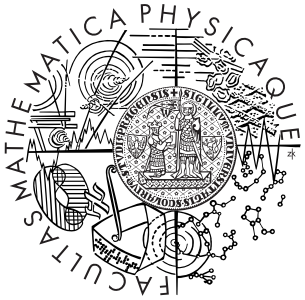
Charles University in Prague
Faculty of Mathematics and Physics

DIPLOMA THESIS



Ivo Pavlík

Pokročilé zobrazování objemových dat na GPU

(Advanced Volume Ray Casting on GPU)

Department of Software and Computer Science Education

Supervisor: Mgr. Lukáš Maršálek

Study Program: Computer Science, Software Systems

2009

First I would like to thank my thesis supervisor, Mgr. Lukáš Maršálek, for his assignment, lots of important advices he gave me and huge amount of time he spent on helping me. He guided me through the whole work process and helped me to avoid many unnecessary problems. I also want to thank RNDr. Josef Pelikán for his willingness to help and for letting me to use his computer with NVIDIA GeForce GTX 280 graphics card for testing. I would like to thank Vladimír Hrinčár for his willingness to help and patient listening to my endless flow of questions. Next, I would like to thank my parents and whole family for their mental support throughout the whole work process. Last but not least I would like to thank all my friends that helped me in any way to finish this work.

I declare I wrote my thesis by myself and listed all used references. I agree with making the thesis publicly available.

Prague, April 17, 2009 Ivo Pavlík

Contents

List of Figures

Listings

List of Tables

Název práce: Pokročilé zobrazování objemových dat na GPU
Autor: Ivo Pavlík
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: Mgr. Lukáš Maršálek
E-mail vedoucího: lukas.marsalek@mff.cuni.cz
Abstrakt: Vizualizace objemových dat, konkrétně vizualizace isoploch nachází uplatnění v mnoha oblastech jako například medicína, antropologie, či vědecká vizualizace obecně. Je to ale metoda výpočetně náročná a dosáhnout interaktivních zobrazovacích rychlostí na běžných procesorech je problematické. Nicméně moderní spotřební grafické akcelerátory (GPU) dávají k dispozici vysoce výkonné, masivně paralelní procesory s dnes už dostatečně obecnými možnostmi jejich programování, díky čemuž představují atraktivní platformu pro uvedené zobrazovací techniky.

V této práci zkoumáme možnosti urychlení vizualizace isoploch založených na metodě vrhání paprsku pomocí oktalového stromu, postavené na systému NVIDIA CUDA. Analyzujeme a vybíráme typ průchodu oktalovým stromem, který je vhodný pro implementaci na GPU a na základě této analýzy implementujeme hybridní průchod objemovými daty, ve kterém kombinujeme bezzásobníkový průchod stromem s přímým průchodem mřížkou objemových dat, založeným na algoritmu DDA. Výslednou metodu jsme integrovali do aplikace WisS pro zobrazování antropologických dat, vyvinuté v diplomové práci [9].

Naše implementace dosahuje až 3,5-násobného zrychlení oproti původnímu DDA algoritmu, které významně zlapšuje interaktivitu aplikace WisS na velkých datových množinách. Na závěr poukazujeme na několik obecných problémů, které obdobné přístupy na GPU musí řešit a navrhujeme směry budoucí práce.
Klíčová slova: objemová vizualizace, isoplocha, GPU, oktalový strom, bezzásobníkový.

Title: Advanced Volume Ray Casting on GPU
Author: Ivo Pavlík
Department: Department of Software and Computer Science Education
Supervisor: Mgr. Lukáš Maršálek
Supervisor's e-mail address: lukas.marsalek@mff.cuni.cz
Abstract: Volume data visualization, namely isosurfaces rendering is of use in many areas like medicine, anthropology or scientific visualization in general. However, it is a compute-demanding technique and the real-time rendering frame rate is hard to achieve on common CPUs. However, the recent consumer graphics accelerators (GPUs) provide a high performance massively-parallel processors with reasonable general programming possibilities and thus present attractive platform for implementing mentioned rendering technique.

In this work we explore the possibilities of the octree-based acceleration of ray tracing-based isosurfacing built on NVIDIA CUDA framework. We analyze and choose the octree traversal type suitable for the GPU implementation, and according to the analysis we implement a hybrid volume traversal combining the stackless variant of octree traversal and direct DDA-based grid traversal. We integrated the implementation into the WisS application – the anthropological data visualization software developed in [9].

Our implementation achieves up to 3.5-times acceleration when compared to the original DDA-based algorithm what results in notably improved the interactivity of the WisS application when used on large datasets. In the end we point on several general problems that analogical approaches on GPU have to solve and we propose possible ways of future work.
Keywords: volume rendering, isosurface, GPU, octree, stackless.

Chapter 1

Introduction

1.1 Motivation

In this thesis, we are interested in and dealing with the area of scientific visualization, specially with acceleration of volume rendering of scalar datasets.

According to Lim and Shin [16] the purpose of *volume rendering* (sometimes referred to as *volume visualization*) is to extract meaningful visual information from a volume data. This approach differs from the surface-based rendering which deals with objects that are represented as surfaces such as polygonal meshes, NURBS patches or subdivision surfaces [26]. The volume data are mostly provided as a 3D grid of volume elements (more often abbreviated as *voxels*) but they can also be extended to more dimensions (e.g. for time dimension in time sequences).

The pioneers of volume visualization developed rendering techniques for visualization of volumetric objects defined by densities like clouds, fog, flames, dust or particle systems [10] but nowadays the volume rendering is of use in many areas of scientific visualization too. For instance *computed tomography* (thereinafter *CT*), *positron emission tomography* (thereinafter *PET*) or *magnetic resonance imaging* (thereinafter *MRI*) are very common sources of volume data used for visualization. But there are also many other scientific applications benefiting from volume rendering whose visualization can be very useful for researchers like computational fluid dynamics (CFD) [29], geological and seismic data, abstract mathematical data such as 3D probability distributions of pseudo random numbers [26], as well as electric fields of molecules to mention a few. While the volume rendering is becoming more interesting also for visual arts and interactive games [26], this work is concerned mostly with scientific applications.

1.2 Volume rendering

There are various methods of volume rendering but each can be described by similar sequence of basic steps that we call *common volume rendering pipeline*. Common volume rendering pipeline was summarized for example in [19] as follows:

1. Obtaining the data
2. Data preprocessing
3. Data classification
4. Projection of the data onto the image plane

5. Displaying the image plane pixels

Obtaining the data In this step the volume data are measured, computed or generated. Visualization methods assume that volume data from this step are processed correctly (e.g. according to Nyquist-Shannon sampling theorem) therefore they can be reconstructed accurately. Very often a scalar data are used for visualization.

Data preprocessing The original volume data may not be immediately suitable for rendering and should be appropriately preprocessed. This can include adjusting of brightness, contrast, gamma, histogram equalization or noise removal. Sometimes conversion to regular grid or data registration (e.g. when combining various data sets) also have to be performed.

Data classification The volume data classification is a process of mapping the volume data values to the values used for visualization. For methods displaying volume data directly this mostly involves setting the color and opacity, for surface-extracting methods this mostly involves setting the isovalue (discussed in detail later). Propagation of the light through the volume can also be precomputed during this step. For the user of a volume rendering application the data classification is the most important step of rendering pipeline.

Projection of the data onto the image plane This is the most specific step of the volume data rendering process and it varies from method to method. In general in this step algorithms decide which voxels and how affect pixels of resulting image on virtual *image plane* (sometimes referred to as a *projection plane*). As foreshown in previous paragraph the volume rendering algorithms can be basically divided into two groups:

- The methods visualizing volume directly, often referred to as *direct volume rendering* (thereinafter *DVR*)
- The methods *extracting a surface* (e.g. into a proxy geometry)

The DVR methods use mapping from a (scalar) volume data to visual properties like color and opacity and then approximate interaction of the mapped data with light. Special subgroup of the DVR are methods displaying isosurfaces only. The *isosurface* is a surface described as points of the (reconstructed) volume with specified value, so-called *isovalue*. Sometimes it is useful to display multiple isosurfaces (for various isovalues) within one volume. Direct rendering methods visualizing isosurfaces are sometimes referred to as *isosurfacing* [22, 21] and we will use this term throughout the thesis as it is more convenient.

Surface-extracting methods displays isosurfaces too but unlike DVR methods they approximate the surface using a proxy geometry which is then rendered.

In this thesis we are dealing with acceleration techniques for isosurfacing.

Displaying the image plane pixels The last step is displaying the result of previous steps. It is typically similar or even the same for all methods and it mainly depends on the architecture the algorithm is implemented on.

1.2.1 Isosurface rendering

Visualization of isosurfaces has traditionally been done by extracting surfaces through a polygonal approximation, usually via some variant of *marching cubes* algorithm (thereinafter *MC*) [17, 20, 29]. The resulting polygonal mesh was then rendered using common graphics hardware. Basic and non-robust variant of this approach is relatively easy to implement and it can provide sufficient interactive rendering speeds however it has several drawbacks. First, the extracted polygonal approximation is tied to one particular isovalue and it has to be recomputed each time the value is changed. This causes problems in applications where user needs to change isovalue interactively. Second, the possibilities of more complex lighting models that can be useful for better visual perception of shapes are limited or not easy to implement [29]. Third, the piecewise linear approximation of volume by polygonal mesh can be topologically different from the actual isosurface in the volume data [1]. Such behavior can be undesirable in some scientific applications. Recently a bezier-approximation approach has been proposed in [11], that can partially improve topological correctness.

Another, more accurate way of rendering isosurfaces, as summarized in [29], is the direct rendering using ray tracing or some form of pre-integrated direct volume rendering. *Ray tracing* (often written *ray-tracing*) is a rendering technique that generates an image of a 3D scene by casting and tracing virtual rays from the camera through the image plane pixels to the scene in order to gain an information about incoming light from the scene. When the ray hits the closest surface, the algorithm computes the light contribution of the intersection point according to the incoming light from the light sources and the surface material properties. One or more recursive rays can be cast from each intersection point to the surrounding scene according to the surface material properties to gather more light information for higher fidelity of rendering. If the ray hits a transparent surface, it can, after executing the above-mentioned computations, iteratively continue the traversing through the scene and gather light information from surfaces behind the hit surface. Lightweight version of ray tracing without recursion is often referred to as *ray casting*. In this work we will use the term ray tracing to keep higher level of generality. More information on the ray tracing can be found for example in Josef Pelikán courses slides on computer graphics at Charles University in Prague (`http://cgg.mff.cuni.cz/~pepca/`).

Since the volume data is built of voxels only, there is no explicit surface information that can be used for a surface-based ray tracing and the isosurfaces need to be explicitly reconstructed.

Compared to the polygonal approximation methods, the ray tracing offers a number of advantages. It can locally interpolate the volume data so that resulting surface will be more accurate. Since the methods based on the ray tracing usually use at least trilinear approximation the resulting surface doesn't suffer from topological ambiguities that appear in MC and ray-tracing methods also naturally support global illumination effects. [29] Finally, ray tracing-based isosurface rendering doesn't need to regenerate isosurface approximation on isovalue change.

Despite all mentioned advantages, the ray tracing algorithms were always considered computationally too demanding for interactive use unless implemented on parallel supercomputers as in [22, 21]. However, recent advances in ray tracing techniques and increasing power of computers caused that it is possible to implement polygonal [30, 28] and even isosurface [29] ray-tracer on consumer hardware with almost interactive frame rates. Despite these advancements, the performance is still not fully satisfactory for many

applications.

1.2.2 Utilizing GPUs

Graphics processing unit (thereinafter *GPU*) is a dedicated graphics rendering device for a personal computer, workstation, or game console. In personal computers it usually sits in on top of a video card or can be integrated directly into the motherboard. [31] GPU original purpose is to efficiently handle specific 2D or 3D graphics-related tasks, thereby freeing CPU computation power for other tasks. Contemporary GPUs are massively parallel special-purpose processors that have overtaken CPUs in terms of *floating point operations per second* (*flops*) [3] which makes the GPUs interesting hardware platform for solving numerous parallel compute-intensive tasks. Ray tracing as both computationally demanding and massively parallel algorithm is one of the first candidates. [7]

There were numerous attempts to employ modern GPUs for purposes of various variants ray tracing algorithms. For instance, Maršálek in 2005 [19] implemented DVR on GPU or in 2007 works [23, 7] on polygonal ray tracing on GPUs appeared. In 2008 paper on ray tracing of general implicit functions on GPU has been presented [12]. There were also some works on isosurface rendering published, e.g. Hrinčár in 2008 [9] or tutorial by Rezk-Salama et al. presented on SIGGRAPH conference the same year [26], but this field is still not yet exhaustively explored.

1.3 Goal

The isosurface ray tracing implementations on GPUs still hasn't reached the top performance and there is probably room for more efficient approaches. The goal of this work is to propose and implement a speed up technique for the ray traversal through the volume data on current GPU architecture and to evaluate its capability to improve the ray tracing-based isosurfacing performance for larger data sets that can be loaded into current consumer graphics accelerators (cca from 512^3 up to 1024^3 voxels). Proposed speed-up technique should be implemented within the *WisS application* implemented by Vladimír Hrinčár in his diploma thesis [9]. The isosurface intersection implementation is not addressed in this work – an appropriate parts of *WisS* implementation will be reused.

1.4 Structure of the thesis

This work is sectioned into several chapters and two appendices are added at the end. Here we briefly describe all of them.

Chapter 2 Here we discuss the previous approaches to classical ray tracing-based isosurfacing on CPU and attempts of GPU implementation of ray tracing-based algorithms and present main idea of our approach – octree-based acceleration structure.

Chapter 3 Here we discuss the characteristics of GPU programming and introduce used programming tools. Next we propose several possibilities of employing chosen data structure and select the tree best candidates.

Chapter 4 Here we analyze candidates in more detail and chose the best one.

Chapter 5 Here we provide implementation decisions and details of our work.

Chapter 6 Here we discuss measurements-based tuning of our implementation and present final results of our implementation.

Chapter 7 In the last chapter we summarize our work and discuss possible ways for future work.

Appendix A Here we provide a brief introduction to the WisS application for new user.

Appendix A.1 Lists the content of appended DVD.

Chapter 2

Previous work

2.1 Ray tracing-based isosurfacing

Ray-tracing based isosurface rendering is not very young technique. One of the first works in this field was published by Levoy in 1988 [14]. Since the time a lot of effort was made to improve the performance of the isosurfacing. Very often, techniques used in classical polygonal ray tracing like empty-space leaping were employed [15]. The principle of empty-space skipping for volumes is illustrated in figure 2.1. The isosurfaces are present only in a fraction of the volume cells. Instead of traversing the volume cell-by-cell (e.g. DDA algorithm), we hierarchically search for the first cell containing the isosurface along each particular ray.

In 2005 Wald et al. [29] presented isosurfacing speed up based on implicit kd-trees. The word *implicit* means that the tree is not built for each selected isovalue but a full tree containing all voxels is used and the tree for the isovalue is built implicitly during the traversal according to information stored in tree nodes. Nodes contain information on what values they contain. The paper uses the *min/max approach* presented in 1992 by Wilhelms and Gelder [32] where only minimal and maximal values contained in node are stored. Storing the values interval appears to be efficient trade off between per-node stored data size and performance gain.

Another approach of improving performance was proposed by Knoll et al. in 2006 [1]. They used implicit octree similar to previously mentioned kd-tree but they also used it for data compression by consolidating cells with same or similar values.

In 2007 Lim and Shin [16] proposed more sophisticated way of empty-space leaping

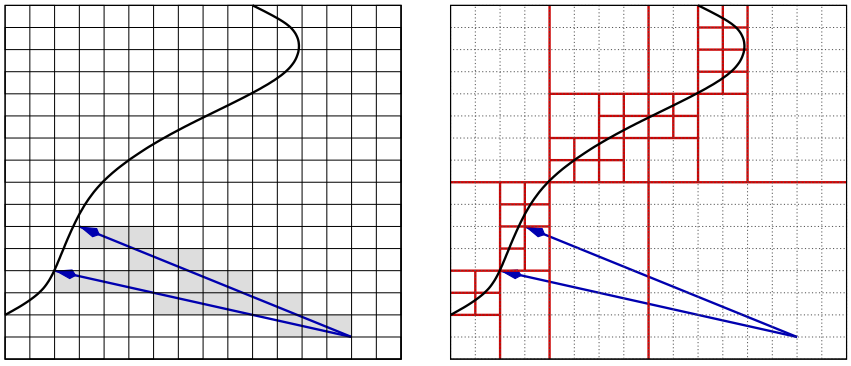


Figure 2.1: The idea of the empty-space skipping acceleration technique. The image reused from [29].

using a new data structure named *half-skewed octree*, which is an auxiliary octree to support the conventional octree.

2.2 Ray tracing-based isosurfacing on GPU

There were several attempts to render isosurfaces via ray tracing on GPU.

For instance, in 2003 Kruger and Westermann [13] implemented multi-pass GPU isosurface renderer. They generated ray by rasterizaton technique and used 3D grid with constant size (8^3) for empty-space skipping acceleration. In 2005 Hong et al. [8] presented ray caster that used object-order approach, where volume sub-cells were rendered subsequently and the whole processing was accelerated using min/max octree.

In 2008 these approaches and many others were discussed in detail in a SIGGRAPH course [26]. Here they proposed some empty space-leaping techniques based on proxy geometry for shader-based GPU isosurfacing. They use fixed-sized grid of blocks (e.g. 4^3 or 8^3) with min/max information attached to them and according to it they determine the "activeness" of each block. Activeness of blocks has to be re-evaluated on each isovalue change. They use two-level adaptive sampling to traverse through the volume. Lower sampling rate is used by default and when the volume values at the current point become too close to the isovalue, the algorithm switches to a higher sampling rate. To guarantee that no intersection is missed the default sampling rate must respect the maximal volume gradient.

Earlier attempts can also be found in Hrinčár's master thesis [9]. We are referring to this thesis because it is recent and because it is a foundation to the WisS application that we are integrating our approach to. He implemented in his work the isosurface renderer that traverses the volume directly voxel by voxel using 3D variant of DDA algorithm. There are used simple speed-up techniques like early-ray termination (i.e. when color buffer's alpha exceeds allowed threshold the traversing of the volume is terminated) but no empty space-leaping technique was implemented there.

A lot of work has been done on other types of ray tracing on GPU that can serve as an inspiration for implementing ray tracing-based isosurfacing on GPU.

2.2.1 Ray tracing on GPU

The polygonal ray tracing is probably the most explored type of ray tracing on GPU. First attempt to implement polygonal ray tracing was done in 2002 by Carr et al. [2]. They implemented hybrid approach where geometry was streamed from CPU and GPU performed only the ray-triangle intersections. The main problem of this approach was the high bandwidth. Purcell et al. [24] tried to solve this problem in a GPU simulator by implementing all necessary task on GPU. They used regular grid as acceleration structure. First implementation on real GPU was made by Purcell in his dissertation thesis [25] in 2004. The remaining problem was still relatively high communication bandwidth and suboptimal acceleration structure [23].

As kd-trees are considered the most efficient speed-up structure for ray tracing of static scenes it is natural that attempts to implement kd-trees on GPUs appeared. One of them was made by Ernst et al. in 2004 [4]. This implementation used multi-pass approach with parallel stack which lead to high kernel-switching overhead and high stack memory consumption. Better result were achieved next year by Foley and Sugerman [6] implementing stackless algorithm *kd-restart* and its modification *kd-backtrack*. Despite

the performance improvement up to factor 2, the implementation still did not outperform CPU raytracers, probably due to high redundancy of traversal steps according to [23]. Stackless approach was improved two years later in 2007 by Popov et al. [23] using kd-tree with *ropes* and *bounding boxes* per leaf for more efficient adjacent leaves location and traversal. This approach provides higher performance at the price of higher memory footprint for the tree.

Several works implementing a bounding volume hierarchy (BVH) for ray tracing acceleration on GPUs also appeared. In 2005 Thrane and Simonsen [27] implemented BVH with stackless traversal that outperformed both kd-restart and kd-backtrack, but was limited due to high bandwidth. Improved BVH implementation performance was achieved by Günther et al. in 2007 [7]. They proposed a parallel packet traversal algorithm using a shared stack with memory-efficient acceleration hierarchy and performance comparable to kd-trees. This algorithm uses less registers and thus can achieve higher GPU occupancy.

Another speed-up structure was proposed in 2008 by Knoll's poster [5] for point clouds rendering. This paper suggested using balanced octree traversed via iterative back-recursion similar to traversal used for general implicit surfaces rendering in [12]. The algorithm benefits from using a few registers, however unlike [12] it uses stack with parents for back-recursing.

2.3 Our approach

Problems of implementing ray tracing mostly stem in inefficient utilizing of the GPU power. Multi-pass approaches consume a lot of processing time and mostly require extra memory for storing temporary results. Standard stack traversals of kd-trees use lots of fast shared GPU memory due to their low branching level and resulting higher depth. This results in low utilization of processing cores. Live registers count is also a problem causing GPU utilization problems.

Our work is trying to address similar problems for volume traversal. Although octrees are not considered the most suitable for polygonal case they seem promising for volume case. Octrees are, thanks to 4-times higher branching level, shallower than kd-trees which can save the fast but scarce GPU shared memory required for stack. Using a full octree the stackless traversal presents itself too. In this work we discuss several GPU-friendly variants of *implicit full octree* traversals and implement the most promising approach using CUDA.

Chapter 3

Octrees and GPUs

This chapter introduces GPU programming using CUDA and reviews possible octree traversal approaches. Sections 3.1.1, 3.1.2 and 3.1.3 are based mainly on [3] unless otherwise stated.

3.1 CUDA programming

CUDA (*Compute Unified Device Architecture*) is the most significant framework for accessing the massive parallel floating-point computational power of contemporary GPUs. It has been developed by NVIDIA Corporation and the beta version of a SDK and a compiler was released in February 2007. CUDA consists of a compiler for slightly modified *C language*, API and software development kit (SDK). It allows programmer to write code that executes on current massively-parallel, multi-core NVIDIA GPUs (GeForce 8 series and higher) directly without the necessity of manipulating of unhandy special-purpose graphics interfaces. Compared to shaders approach, the CUDA-based kernels have greater programmatic flexibility while the programmer can still employ some specific graphics features like texture fetching.

Other API similar to CUDA also appeared on the scene like *Stream* SDK on AMD FireStream processors (previously called ATI Close To Metal) but it never came very popular and is now deprecated. Recently (at the end of 2008) new open source framework called *OpenCL* for programming on heterogeneous platforms including parallel GPU architecture appeared. This API will be probably supported by both ATI and NVIDIA GPUs but it will take some time for it to run in.

3.1.1 CUDA architecture

This section provides a brief introduction to the CUDA architecture.

A CUDA programmer is provided a powerful parallel programming model based on few abstractions that helps to use capabilities of modern NVIDIA GPUs simply and efficiently. The main abstraction of the CUDA model is a hierarchy of thread groups that are executed independently. It guides a programmer to partition problem into independent sub-problems and to smaller pieces that can be solved cooperatively in parallel which provides a transparent scalability of code without the need of exact knowledge of current GPU capabilities. Moreover it offers efficient, fine-grained parallelism that allows to decompose solved problem at data element level.

Each thread executes the same function called a *kernel* (like in the stream processing paradigm). The kernel is defined using `__global__` specifier and a group of threads

```
// Kernel definition
__global__ void myKernel(float* in, float* out)
{
   // Do the computing of one thread
   //...
}

int main()
{
   // Kernel invocation
   myKernel<<<1, N>>>(in, out);
}
```

Listing 3.1: Simple example of CUDA program with empty kernel and kernel invocation in one one-dimensional block of threads.

executing the kernel is invoked by a newly introduced syntax `<<<...>>>` as shown in the example 3.1.

Threads are executed in groups called *thread blocks* (see figure 3.1) that can be one-, two- or three-dimensional. Threads within a block are assigned an unique ID so that each thread can identify itself to work on its part of stream data. Threads within a block can communicate together using per-block *shared memory* and built-in synchronization function `__syncthreads()` which does the "rendezvous synchronization" of all threads in the block. Furthermore multiple equal-shaped blocks can be executed in so-called *grids*. Grid of thread blocks can be one- or two-dimensional and, similarly to threads within one block, each block is assigned unique block ID. Total count of executed threads per one `<<<...>>>` execution equals threads count per one thread block times number of thread blocks per grid. The order of blocks' execution is undefined, blocks are thus required to be executable in any order or in parallel (i.e. must not depend on each other). The dimension of the grid together with the dimension of the blocks is referred to as the *execution configuration*. Actually the execution configuration contains a bit more information but it is not important for us now.

Important restriction on kernels worth mentioning is that they cannot use the *recursive calls* because GPUs don't implement function calling stack and all functions are inlined into the one kernel function.

The CUDA programming model is not dedicated to GPU architecture only. It more generally assumes that threads can execute on separate *device* that serves as coprocessor for *host* that executes main C program. Both device and host are assumed to have their own memory (*host memory* or *device memory* respectively). Host can access the device memory through the provided API calls only.

Threads can use more kinds of memory of different characteristics (see figure 3.2). All threads from all blocks can access one common *global memory*. Each thread has its own *local memory* that is used for some automatic kernel variables. The local memory is actually part of the global memory but each kernel can access only its own instance of memory (e.g. local variable). Each block of threads has its own so-called *shared memory* accessible by all threads of the block. Local, shared and global memories allow read-write access. There are another two types of memory available: *constant memory* and *texture memory*. Both types are read-only and both can be accessed from all threads. Texture memory also offer more specific functionality like filtering and special addressing modes.

Each memory type is optimized for a different usage. Global memory is not cached and accesses to this type of memory have to follow right pattern to achieve optimal memory

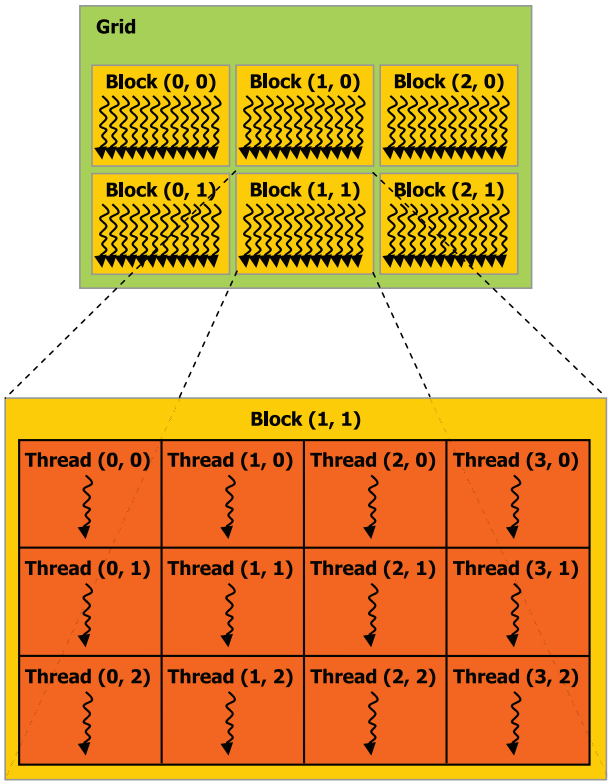


Figure 3.1: Grid of thread blocks. The image reused from [3].

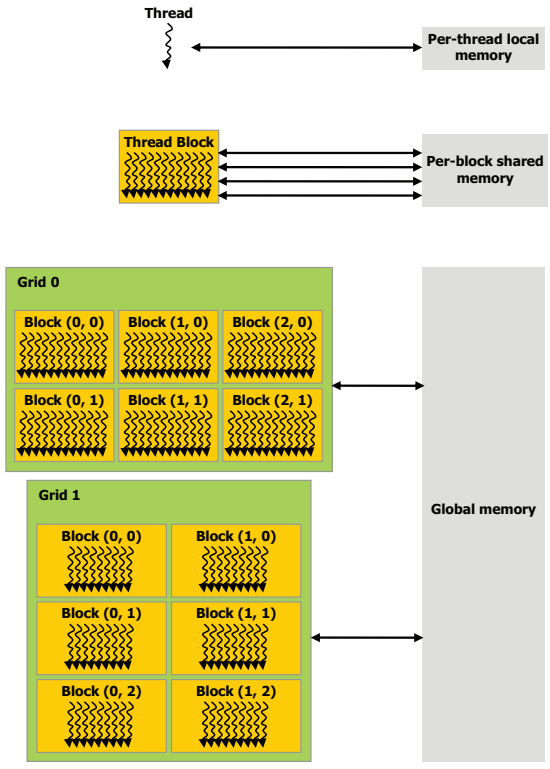


Figure 3.2: CUDA memory hierarchy. The image reused from [3].

bandwidth. For example some types can be loaded into registers in one instruction if they are address-aligned. When accessing the memory by group of threads, accesses can be coalesced into one, and thus efficient memory operation if they satisfy some requirements. Fortunately these requirements are less restrictive on newer generations of NVIDIA GPUs. The shared memory is an on-chip memory so it is much faster than the global memory. Under some conditions, i.e. when there are no access conflicts between threads (so-called *bank conflicts*) using the shared memory can be as fast as accessing registers. It is common technique to load often accessed data from the global memory to the shared memory, work with the data in the shared memory and finally save results back to the global memory. The texture memory and the constant memory are cached and optimized for reading. Reading the constant or the texture memory location is as fast as reading a register if the value is cached otherwise it costs one read from the global memory. Texture cache is optimized for 2D spatial locality.

3.1.2 Tesla architecture

As mentioned in section 3.1.1, CUDA is designed as general API but only GPU implementation of CUDA is important for this work.

Current NVIDIA GPUs are based on NVIDIA *Tesla* architecture that is built around a scalable array of massively multi-threaded streaming processors, so-called *multiprocessors*. Threads from one block of threads execute on just one multiprocessor concurrently and if there are enough computing resources available (e.g. shared memory, registers) the multiprocessor can even execute more blocks at once. Each multiprocessor has its own shared memory and eight scalar processors with zero thread-scheduling overhead.

Tesla architecture introduced a new architecture called *single instruction, multiple thread* (hereinafter *SIMT*). In this model each thread is assigned to one scalar processor and the SIMT mechanism manages and executes groups of 32 parallel threads called *warps*. Threads in warp are free to branch independently but because multiprocessors can execute only *one common instruction* at a time some performance penalties occur if execution paths of threads in one warp diverge. In that case the execution of warp threads is serialized, i.e. multiprocessor executes each active path of kernel separately while threads that are not on that path are disabled. After execution of all diverged paths the execution continues again in one common path.

For overview of Tesla chip organization see figure 3.3.

3.1.3 Architecture characteristics

Although CUDA can provide an order of magnitude speed up of several kinds of applications, it is obvious that using CUDA will not have desired effect for some applications. To achieve reasonable speed up by utilizing CUDA an application have to comply few requirements.

Applications have to contain parts that are or can be converted into a highly data-parallel computation. It means that the same part of program is executed on many data elements in parallel. Requirement that improves efficiency of CUDA implementations is a large data set to process and high *arithmetic intensity*[1] of computation. When both requirements are met, expensive device memory accesses can be interlaced with arithmetic calculations by the thread scheduler allowing application to employ computing power of underlying GPU more efficiently.

[1]The ratio of the arithmetic operations count to memory operations.

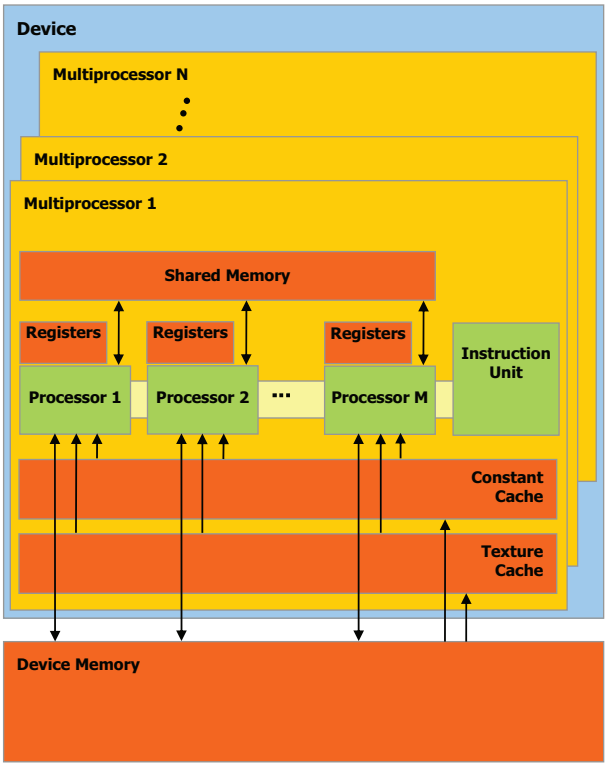


Figure 3.3: Tesla chip organization. The image reused from [3].

A good estimation of kernels ability to work efficiently is the so-called *multiprocessor occupancy* (or just *occupancy*). The multiprocessor occupancy is defined as the ratio of the number of active warps per multiprocessor to the maximum number of active warps on given GPU. The occupancy is determined by the amount of computing resources required for given kernel and it's configuration, i.e. the registers count used by thread, the shared memory amount needed per block and the threads count per block. For example, the more registers are needed per thread or the more shared memory is needed per block, the less threads can be active on one multiprocessor and the smaller the occupancy will be and therefore worse kernel behavior (e.g. memory latency hiding) can be expected.

Another worth-mentioning characteristic influencing performance is between-threads data dependency. Data dependent threads need to be synchronized which consumes extra processor cycles.

Ray tracing algorithms are naturally parallel because each ray can be processed independently without the need of mutual communication or synchronizing with others ray computations. The problem of such algorithms is that arithmetic intensity is not very high due to frequent access to data structures (e.g. scene or volume data, acceleration structure).

Computing abilities of CUDA-enabled NVIDIA GPUs are described by so-called *compute capability*. The compute capability is described by two-digit number and it tells how much shared memory and how many registers are available per block, how many active warps can a multiprocessor handle, whether there is a native support for 64-bit double operations and many others. Current GPUs are of capability from 1.0 to 1.3 and for our analysis we always consider having GPU with currently higher available capability 1.3.

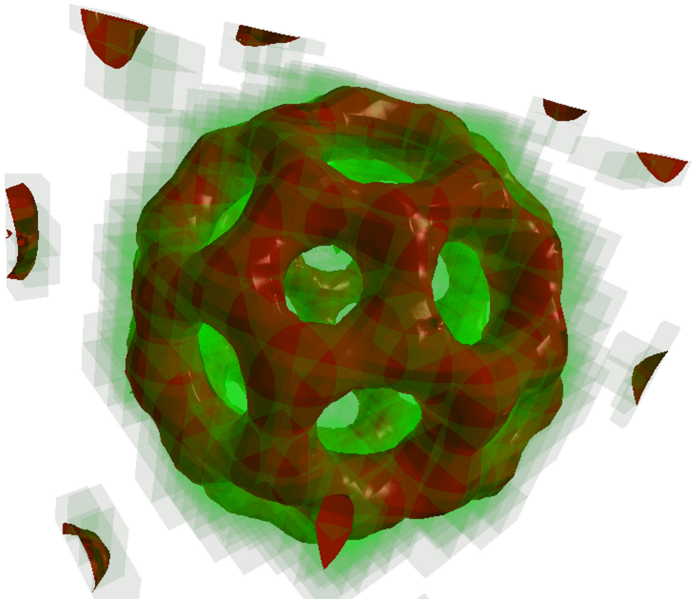


Figure 3.4: Visualization of 2^3 macro cells on *Bucky* dataset.

3.2 Octree traversal

In this work we want to implement an octree acceleration that allows changing isovalues without dramatic performance overhead while still speeding up volume traversal.

We use implicit octree with min/max values similar to tree structures used in [32, 29, 1] – each node contains minimum and maximum value of whole node subtree. This approach requires no computations on isovalue changes since the tree is built implicitly during the traversal.

We decided to use full octree with size s^3, where s is the smallest power of 2 greater than or equal to all sides of visualized volume data. This approach can be memory-consuming (e.g. when used on irregular-shaped volumes) however allows straightforward stackless traversal implementation since addresses of node's child or parent can be simply computed on the fly and thus don't need to be stored in the tree. The octree is stored separately from the volume itself and contains the min/max values only.

We don't build the tree down to voxels because the tree would be few-times larger than the volume data. We utilize the concept of *macro cells* instead. A macro cell is represented by one tree leaf and it corresponds to a s^2-sized regular segments of volume data (e.g. cells with size $2^2 = 4$ in one dimension). Volume is then traversed in *a hybrid* way where first the tree is traversed using the technique discussed below and after reaching the leaf, the corresponding macro cell is traversed using plain 3D DDA algorithm with isosurface intersection detection. This approach also leave some room for experiments with macro-cell size. A visualization of macro cells with size 2^3 on standard NVIDIA Bucky dataset is shown in figure 3.4.

In our work we describe the size of a macro cell is described as levels count of a sub-tree replaced by the macro cell. For example the macro cell with size 4^3 would replace sub-tree with 3 levels and thus the macro cell is described as 3 level macro cell. Similarly a 1 level macro cell corresponds to subtree with 1 level and thus it corresponds to one voxel.

3.2.1 Possible traversal approaches

This section briefly discuss several possibilities of traversing an octree from the traditional full-stack traversal through reduced-stack traversals to a completely stackless approach. Each traversal type provides different trade-off between the stack size and the amount of necessary computations. The most appropriate approaches are selected the discussion.

Since there is no native runtime recursion available at GPUs, a stack of recursive calls must be handled manually using explicit stack for storing to-be-processed items for each thread. If a (parallel[2]) stack is used, it have to be stored in the shared memory because it is accessed often and storing it in global memory would be very expensive. The size of the stack have to be fixed thus the depth of the octree is limited. Since current GPU can only load datasets up to cca 1 GB we have set the dataset size limit to 1024 voxels per dimension. The maximum depth of an octree without macro cells for such a data set won't be taller than 11 levels. Taking into account that macro cells should be at least 2 levels large (to be larger than a voxel) the maximum depth of an octree with macro cells won't be greater than $11 - (2 - 1) = 10$ levels. The stack of each stack-using approach must be sized to this depth.

Full index stack traversal This is the classical tree traversal approach where *full coordinates* of to-be-processed nodes are stored on the stack. When a node is being processed first to-be-traversed children are identified. If there are more such children, all of them except the first one are stored on the stack and the first one is processed right in the next algorithm cycle. If there is no node to process from the previous cycle the next node is popped from the stack. If the stack is empty the algorithm ends.

The list of children intersected by the ray is detected by intersecting the cell's mid-planes. Sorted intersections that are inside the cell boundaries determines which bit changes in the relative coordinates of children when traversing through the cell along the ray.

Each octree node can produce up to tree children that have to be stored on the stack. Since we do not store the root on the stack and we use 32-bit coordinates for nodes, the size of the stack will be $9 \times 3 \times 32 = 864$ bit (108 B).

Parent-storing stack traversal This approach exploits the redundancy of node's and it's children's coordinates. Instead of storing separated nodes it stores full coordinates of processed node and small information that list of to-be-processed children can be computed from. Until all children are processed for the current node the record remains on the stack. After processing of all children from the record, whole record is deleted from the stack. There are at least two variants of this approach.

Children storing variant This variant saves for each processed node a record containing the full coordinates of this node and relative 3-bit coordinates of its children identified for processing. A 3-bit mask of already processed children is also stored within each record and adjusted after popping each children value.

Each stack record contains the full 32-bit (parent) node coordinates, three relative 3-bit coordinates of children and one 3-bit mask of not-yet-processed children, thus taking $32 + 3 \times 3 + 3 = 44$ bits (6 B). Since we don't need to store records for leaves (they do not have children to process) the size of the stack will be $9 \times 6 = 54$ B.

[2]In this thesis we do not consider the approaches with stack shared between more threads.

Mid-planes storing variant This variant differs from the previous one by postponing the computation of children address to the time of popping the child from the stack. During the node processing only the intersected mid-planes of the cell are stored as 2-bit indices with 2-bit local-stack index. Mid-planes' indices are stored in reversed order and local stack index says which of stored mid-planes is next to be processed (e.g. if it is equal to 0, there is no unprocessed intersected mid-plane).

Each stack record contains full 32-bit (parent) node coordinates, three 2-bit indices of mid-planes and one 2-bit local stack index, thus taking $32 + 3 \times 2 + 2 = 40$ bits (5 B). Similarly to the previous case, we don't need to store records for leaves and thus we save one level and the size of the stack will be $9 \times 5 = 45$ B.

Skip-storing stack traversal This traversal approach is based on parent-storing stack traversal. The difference between these approaches is that skip-storing stack traversal saves stack size by replacing the full node coordinates by so-called *skip counter*. The skip counter stores count of levels skipped from the previously stored stack record or from the root if there is no such a record. This information allows to recompute the omitted full node address when moving up the tree. 4 bits are sufficient for storing all possible skips in tree 10 levels deep. Similar to parent-storing stack traversal there are several variants of this approach.

Full variant This variant behaves very similar to the parent-storing stack traversal – children storing variant. It's stack record consist of the 4-bit skip counter, three relative 3-bit coordinates of children and one 3-bit mask of not-yet-processed children, thus taking $4 + 3 \times 3 + 3 = 16$ bits (2 B). Since we don't need to store records for leaves the size of the stack will be $9 \times 2 = 18$ B.

Counter-and-mask variant This variant differs from the previous one in omitting the children relative addresses. Each time when moving to next child of current (parent) node, mid-planes must be intersected with the ray. According to already traversed children mask and mid-planes intersections the next children coordinates are computed.

Each stack record contains 4-bit skip counter and one 3-bit mask of not-yet-processed children, thus taking $4 + 3 = 7$ bits (1 B). Since we don't need to store records for leaves the size of the stack will be 9 B.

Counter-only variant This is the most lightweight variant of the skip-storing approach. The stack record contains the 4-bit skip counter only. Both parent and next child must be recomputed each time the node is entered during up-traversing the tree. Two stack records can be stored in one 1 B variable therefore whole stack stored in five such variables (e.g. in array) will take 5 B of shared memory.

Stackless traversal Stackless traversal is very similar to the skip-storing stack traversal – counter-only variant. The main difference is that the stackless traversal doesn't even store the skip counter and behaves like the skip counter was saved at each down-traversed tree level (i.e. was always equal to 1). When up-traversing the tree the current (parent) node and next child coordinates need to be computed. Since there is no information stored per processed node no stack is needed.

3.2.2 The best candidates

From this list of approaches it is necessary to pick the shortlist for more detailed analysis. As the main criteria for this first selection we use the multiprocessor occupancy behavior for selected GPU compute capability as well as the necessary computations complexity estimate. We expect to have 32 registers available for occupancy estimation because it is not very likely to make our implementation go below this count. Various thread counts are examined when searching for the best achievable multiprocessor occupancy. Other characteristics like accessing the tree in memory are not considered because they are performed in each type of traversal.

As the traversal using the full index stack requires 108 B per stack (what leads to huge amount of memory per block), it can be immediately disqualified because it cannot beat the 13 % occupancy level at any reasonable configuration.

Parent-storing approaches uses markedly less memory for the stack than the full stack approach, and thus they are able to achieve better occupancies. Although the children-storing variant can achieve 25 % occupancy it is still quite low value. The mid-planes-storing variant due to better stack compression achieve 31 % occupancy. The latter is even more efficient in computation of the next children during up-traversing phase. It intersects the proper mid-plane according to the information from the stack instead of intersecting all mid-planes and searching for first next child according to the mask. Because of this the children-storing variant can be disqualified.

All three skip-storing approaches require sufficiently small stack and therefore they are not limited by used shared memory size and can achieve 50 % occupancy for 32 registers and several thread count configurations per block. Although the register usage become the main occupancy limitation we prefer the approaches saving more shared memory that can be used for other purposes (e.g. caching a frequently used data).

While the full variant on up-traverse needs to compute the coordinates of current (parent) node and do the simple computation of the full children coordinates from the relative coordinates stored on the stack, the next two variants need to do more work. The counter-and-mask variant needs to recompute ray-mid-planes intersections for a next-traversed child. The counter-only variant needs to find the closest intersection in a similar manner. Since the mask stored in the counter-and-mask variant doesn't make the computation much more efficient and the latter approach saves more shared more memory we disqualify the the counter-and-mask variant. Since the memory savings of counter-only variant is almost 3-times higher than the full variant we decided to discard the full variant too.

The stackless traversal can achieve 50 % occupancy for 32 registers similarly to skip-storing traversals. It requires computations similar to the counter-only variant. The difference is that the stackless approach cannot skip levels during up-traversal like the stack-based approaches do, but on the other hand the lack of stack manipulation (accessing the memory and packing/unpacking the stack records) can slightly improve the traversal performance. Due to this we decided to leave the stackless traversal for more detailed analysis too.

Chapter 4

Reduced-stack traversal approaches

In the previous chapter we selected three approaches we found potentially suitable for our purposes:

- the mid-planes variant of parent-storing stack-based traversal,
- the counter-only variant of skip-storing stack-based traversal and
- the stackless traversal.

In this chapter we present a closer analysis of these traversals and choose the most appropriate one for implementation.

All three approaches have few steps in common that we will not discuss during the analysis, only the differences between the approaches will be discussed. For instance all approaches uses the same ray-bounding box initialization or resulting color buffer processing. There are also several global variables[1] common to all approaches but only the specific ones will be mentioned. The examples of common globals are the ray structure or the current level and coordinates. Implementation details will be discussed in more detail in the implementation chapter after the best candidate is picked.

4.1 Mid-planes variant of parent-storing stack-based traversal

This traversal distinguishes three different events during it's main cycle:

- a macro cell is reached,
- a node is entered for the first time and
- a node is entered from a lower level.

When a macro cell is reached the common routine is called and it executes the whole DDA grid traversal with isosurface detection and shading. When the function finishes, the algorithm moves upwards to a node according to the current stack record. When entering a node for the first time (from the parent), intersections with mid-planes are computed, the algorithm moves to the first intersected child and intersections are stored on the stack. Then we start traversing through the children of the node until the isovalue

[1] Variables live during the whole main cycle.

```
while (true)
{
   if (level >= levelsCount - 1)
   {
      // Process a macro-cell
      IntersectGrid();
      if (!MoveUpwardsPS())
         break;
      else
         continue;
   }
   else if (traversalDir == dirDown)
   {
      // Enter the node for the first time
      EnterNodePSDown();
   }
   else
   {
      // Re-enter the node from below
      if (!EnterNodePSUp();)
         if (!MoveUpwardsPS())
            break;
         else
            continue;
   }

   if (!MoveToFirstNonEmptyChildPS())
   {
      if (!MoveUpwardsPS())
         break;
   }
}
```

Listing 4.1: Main cycle of the mid-planes variant of parent-storing stack-based traversal in pseudocode.

is not outside the child's min/max range. When entering the node from a lower level the next child is determined according to the next intersected mid-plane from the stack. If there is no next child the algorithm moves upwards to node stored on the stack otherwise the same traversing through the children as in previous case follows. The outline of the algorithm is shown in the pseudocode 4.1 (parent-storing-specific functions use postfix "PS").

Using the same `MoveToFirstNonEmptyChildPS()` function for both second and third case may seem little inefficient for the second case because it may lead to reading from the stack just after the writing when the first child is skipped but we made this decision due to branch shortening. If we used specialized version of this function for each case their branches would be much longer which may lead to higher performance penalties due to thread's execution paths divergence. Our approach allows the threads paths to join sooner when traversing the tree.

The traversal used two specific global variables:

- the stack index and
- the direction of traversal.

4.2 Counter-only variant of skip-storing stack-based traversal

This traversal distinguishes the same three events during it's main cycle as the previous approach:

- a macro cell is reached,
- a node is entered for the first time and
- a node is entered from a lower level.

The macro cell processing is the same as in the previous approach – the common routine is called and it does the grid traversal with the isosurface intersection. When the function finishes the grid traversal it moves upwards. If the current level has its skip counter stored on the stack this value is used for up-traversing otherwise the current skip counter value is used and reset to zero after the usage. Computation of the new node coordinates is done just by right shifting of each dimension by the skip counter. When entering a node for the first time (from the parent) intersections of the ray with node's mid-planes are computed. If there is at least one valid intersection, the skip counter is stored on the stack and reset to zero. The algorithm then moves to the first intersected child and the traversing through the children of the node continues until the isovalue is not outside the child's min/max range. Entering the node from a lower level is almost identical to the previous case. The difference is that the skip counter is not saved to the stack again and that the algorithm moves upwards if there is no next child. The outline of the algorithm is shown in the pseudocode 4.2 on the next page (skip-storing-specific functions use postfix "SS").

Moving the functions `MoveToFirstNonEmptyChildSS()` and `MoveUpwardsSS()` to the end of the cycle helps to reduce the overhead of the worst case execution paths divergence.

The traversal used five specific global variables:

- the stack index,
- the direction of traversal,
- the skip counter,
- flag telling whether the skip counter of the current level is saved on the stack and
- the child level coordinates.

4.3 Stackless traversal

This traversal distinguishes only two events during it's main cycle:

- a macro cell is reached and
- a node is entered.

The macro cell processing is the same as in the previous approaches. It only differs in way of moving upwards – since there is neither next to-process node nor skip counter stored on the stack, the algorithm have to move exactly one level upwards or finish in case the

```
while (true)
{
   childLevelCoords == -1; // Coordinates of next child
   if (level >= levelsCount - 1)
   {
      // Process a macro-cell
      IntersectGrid();
   }
   else if (traversalDir == dirDown)
   {
      // Enter a node for the first time
      isLevelOnStack = EnterNodeSSDown();
      childLevelCoords = GetChildFromT(currentT);
   }
   else if (tCurrent < tOutCell)
   {
      // Re-enter a node from below
      EnterNodeSSUp();
      childLevelCoords = GetChildFromT(currentT);
   }

   if (childLevelCoords != -1)
      MoveToFirstNonEmptyChildSS();
   if (childLevelCoords == -1)
      if (!MoveUpwardsSS())
         break;
}
```

Listing 4.2: Main cycle of counter-only variant of skip-storing stack-based traversal in pseudocode.

```
while (true)
{
   if (level >= levelsCount - 1)
   {
      // Process a macro-cell
      IntersectGrid();
   }
   else if (tCurrent < tOutCell)
   {
      // Enter a node
      EnterNodeSL();
      if (MoveToFirstNonEmptyChildSL())
         continue;
   }

   if (!MoveUpwardsSL())
      break;
}
```

Listing 4.3: Main cycle of stackless traversal in pseudocode.

current node is the root of the tree. Moving to a parent of the node is performed via right shifting of each dimension of the node coordinates. When entering the node, the ray is intersected with appropriate cell's mid-planes and the next child after the current position is determined according to the intersections. After determining the next child the search for the first child containing the isovalue starts. If no such child exists the algorithm moves upwards to a parent of the node. The outline of the algorithm is shown in the pseudocode 4.3 (parent-storing-specific functions use postfix "SL").

This approach doesn't use any specific globals variables.

4.4 Comparison

The basic idea of all three approaches is very similar but there are several details that they differ at. Here we compare the approaches and choose the most appropriate one for the implementation.

The *parent-storing stack-based traversal* can achieve 31 % occupancy due to relatively large stack. Next it uses two more global variables that increase the required registry count by two new registers. This is not very dramatic when compared to twenty one registers occupied by the common global variables but it should be taken into account. More important characteristics than the extra registry requirement is the existence of three significant branches. The more long branches the algorithm can diverge on, the higher performance penalties can occur on SIMT architecture as already mentioned in section 4.1.

The *skip-storing stack-based traversal* can achieve higher occupancy of 50 % due to a very small stack and thus the occupancy is limited just by registers usage. Unfortunately this approach uses five more global variables in our design, increasing the usage by seven new registers (twenty eight register for all global variables). This will result in worse occupancy or more significant explicit limitation of register usage will be required which will force the compiler to save some variables to slow local memory and the performance penalties will occur. The traversal has also the same branching problem as the previous approach. Although we tried to minimize the length of three main branches they are still

present in the design and are all significantly long. There is also a bit more computation required during traversing through the children when compared to the previous approach – all mid-planes must be re-intersected for each next child instead of one mid-plane.

Similar to the skip-storing approach the *stackless traversal* can achieve 50 % occupancy that is limited only by assumed 32 registers requirement. As there are no special global variables, no extra registers are needed during main cycle run for them. Computations are very similar to those in the skip-storing approach but since there is no stack the stackless traversal doesn't access the stack memory and doesn't perform any stack record manipulation (i.e. packing or unpacking the stack records). On the other hand, this is achieved at the expense of not skipping nodes that contain only one intersected child during the up-traversal. According to our empirical evidence for some mutual configurations of the ray origin and the cell, the ratio of number of cases where just one child is intersected to all node intersections count can reach 40 %. However we expect the average ratio for all configurations to be less than 20 % that is tolerable according to our opinion. An important feature of this traversal is that there are only two main branches in the main cycle and thus performance penalties caused by the execution path divergence are not that dramatic as in two previous approaches. The last important characteristics of the stackless traversal is that it has the most straightforward implementation that allows better understanding for the others as well as open room for more obvious optimizations.

4.4.1 Summary

The parent-storing stack-based traversal achieves bad occupancy and needs three main code branches. The skip-storing stack-based traversal requires lot of global variables and has three main branches as well. Although the stackless traversal doesn't skip nodes with one intersected child on up-traversal, it requires no extra global variables, performs no stack accessing and manipulation, has only two main branches and the last but not least it has the most clear implementation. After considering all these characteristics we decided to implement the *stackless traversal* as our acceleration approach.

Chapter 5

Stackless traversal implementation

In this chapter we explain the implementation details of our work. First we introduce the environment that our work is implemented within, then we explain how our main data structure is designed and built and finally we explain in detail how the tree traversal works.

5.1 Environment

Our work is implemented within the *WisS* application programmed by Vladimír Hrinčár in his diploma thesis [9]. This application is primarily intended for volume data visualization in anthropology but it can be used for other volume data sources too. WisS offers 3 basic types of volume data visualization:

- simple transparent isosurface rendering,
- opaque isosurface rendering with shadows and
- direct volume rendering.

WisS application supports rendering of two datasets at a time that is useful for instance when exploring the topological differences of two skulls. In our work we implemented an acceleration technique for transparent isosurface rendering of single dataset as a new rendering mode in this application. The double dataset rendering ability is ignored in this mode. We also added some convenience functionality like configuration files containing raw dataset properties and rendering settings (e.g. isovalue, alpha value) and fixed some bugs (e.g. incorrect alpha integration for transparent isosurfaces, etc.).

The WisS application is basically a .NET application written in C# programming language but almost whole rendering functionality is implemented in a dynamically linked CUDA library called *CUDA Renderer Library* (`CULib.dll`) accessible to the main .NET application through the API defined in the library. There are a few original source code files of rendering library that needs to be mentioned:

- `volumeRender.cu` – this file contains the whole renderer functionality executed on the host (CPU) like loading the volume data from a file and the code launching kernels executed on the device (GPU).
- `volumeRender_kernel.cu` – this file contains hardwired kernel settings, kernel data structures definitions and basic kernel routines.

- `kernel8.cu`, `kernel16.cu` – these files contain kernels implementation for 8-bit datasets or datasets up to 16 bit respectively.

We added the host functionality directly into the `volumeRender.cu` file. After making the decision to employ CUDA 2.1 support for C++ templates we added two more files containing our host code:

- `kernel_basics.cu` – this file contains original kernel code for transparent iso-surface rendering made-over into templates and adjusted for utilization within our octree acceleration code.
- `kernel_trees.cu` – this file contains implementation of our octree acceleration technique.

5.2 Octree data structure

Our acceleration structure is a implicit full split-in-the-middle octree with the size of all dimensions equal to the power of two and same in each dimension. Each *node* of the tree represents a volume *cell*, specially the leaves represent the *macro cells*.

The size of the tree is set according to the size of the dataset as the smallest power of two greater than or equal to the maximum of the sizes of dataset through all dimensions. This implies that the tree range can lap over the dataset range, and thus the tree covers "empty" space that will never be used. The size of a macro cell is a hardwired constant value and is expressed as the number of levels of the tree built down to the voxels covered by macro cells. The size of a macro cell can be computed as

$$2^{macro\,cell\,levels-1}$$

(e.g. macro cell for 3 levels is a cube with side $2^{3-1} = 4$). The size of the cell represented by the inner tree node is determined by the node's level and can be computed as

$$2^{full\,tree\,depth-1-node\,depth}$$

The memory footprint of the whole tree can be simply computed from the nodes count of the tree. The count of nodes of the full octree is equal to

$$\sum_{i=0}^{d-1} 8^i = \frac{8^d - 1}{8 - 1} = \frac{8^d - 1}{7}$$

where d is the depth of the tree. Since we store two values (min and max) in each node the size of whole tree will be

$$2 \times \frac{8^d - 1}{7} \times value\,size$$

where the *value size* expresses bytes count needed to store one volume value and thus one min or max value. For example, the tree for 8-bit 1024^3 dataset with macro cells corresponding to three levels of the tree will occupy $2 \times \frac{8^{11-3+1}-1}{7} \times 1 = 38347922$ bytes (cca 37 MB) of memory. If we reduce macro cells size by on level the size of the tree will jump cca eight times up to approximately 290 MB.

5.2.1 Memory representation

We store our octree in texture instead of the global memory since there is the potential of texture caches, the addressing calculations are hidden better and no special access patterns are required to get good memory performance [3]. There are two ways of bounding data to a texture. First, data can be bound to a texture directly from the so-called *linear memory*, what is in fact the directly accessible global memory. Textures bound this way are restricted to be one-dimensional and no filtering or special addressing modes (e.g. normalized coordinates) are supported. The second way is to allocate the data as a *CUDA array*. The CUDA array is a opaque memory optimized for the texture fetching that can be written from the host only. Textures bound to the CUDA array memory can be up to three-dimensional and support several filtering and addressing modes.

We first decided to store the tree as one-dimensional array in one-dimensional linear memory texture. We didn't use one-dimensional CUDA array texture because it has strong limitations on texture maximum width that can be 2^{13} instead of 2^{27}in linear memory array. This allowed us to store whole 9 levels of the tree to one texture.

We addressed each node of the tree using so-called *linear index*. The linear index is composed of two components. The first component is the *level offset* that tells how many nodes are stored in all levels of the tree that are above the level of the node. The level offset is computed simply as nodes count of a tree with levels count equal to node's level:

$$level\,offset = \frac{8^{level} - 1}{7}$$

The second component is called the *level index* that indexes all nodes within one level. The level index is computed as

$$level\,index = s^2 * z + s * y + x$$

were s is the cell count per one dimension on node's level and x, y and z are level coordinates of the cell represented by the node. The linear index is then computed as follows

$$linear\,index = level\,offset + level\,index$$

Although this approach worked well on data up to 1024^3 for trees with macro cells equivalent to three or more tree levels, we discovered during preliminary testing that there should be possibility of smaller macro cells as well. Unfortunately the range of linear memory was not sufficient for this. After this discovery we decided to slightly adjust the linear index concept by simple remapping to two-dimensional array. We then stored the octree data in two-dimensional CUDA array. Because the range of 2D CUDA array is $2^{16} \times 2^{15}$ the possible range of the octree storage increased $2^3 = 8$-times that gave us the possibility of storing one more tree level. The *2D index* (as we named the remapped linear index) is computed as follows

$$2D\,index = \begin{pmatrix} linear\,index\,\%\,k \\ linear\,index\,/\,k \end{pmatrix}$$

where k is the width of the 2D texture, % represents the remainder for integer division and / stands for integer division. We set k as the power of two that is closest to the square root of the nodes count

$$k = \min\left\{2^i : i \in \mathbb{N}, 2^i \geq \left\lceil \sqrt{nodes\,count} \right\rceil \right\}$$

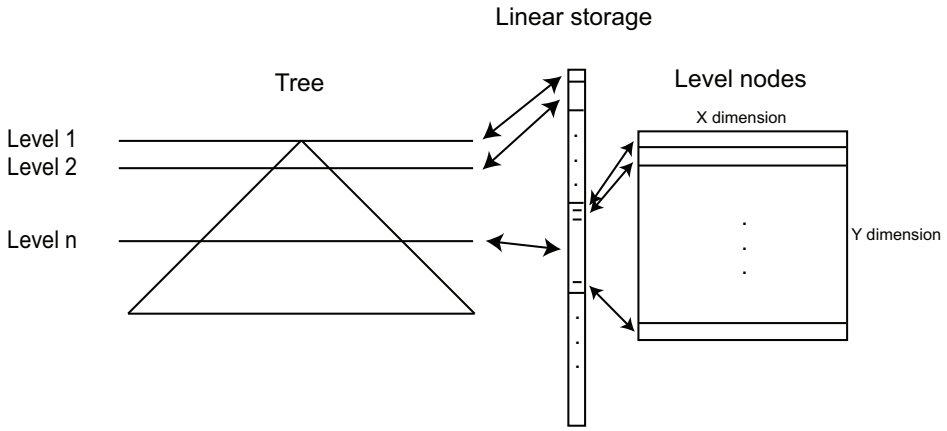


Figure 5.1: Octree linear indexing.

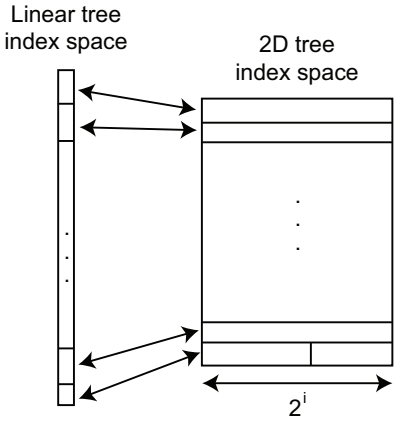


Figure 5.2: Mapping of linear octree index to 2D octree index.

Choosing the width being the power of two allows us to use efficient mapping implementation using bitwise operators. The measurements of both linear and two-dimensional approaches shown that using the new two-dimensional index causes no performance penalty.

We store each node of the tree in one vector variable (`uchar2` for 8-bit data, `ushort2` for 16-bit data). This way we have two time less texels count and possible better usage of bus bandwidth since the whole node is transferred during one memory operation instead of two. We use value normalization to **`float`** type because the isovalue used in the WisS implementation is of **`float`** type. The texture of course doesn't use any special filtering mode since using any kind of filter different from the box filter (nearest neighbor) makes no sense in this situation.

The tree linear tree index mapping and 2D tree index mapping ideas are depicted in figures 5.1 and 5.2 respectively.

5.3 Octree build

The tree is built in two phases.

Leaves initialization This phase is performed during the data loading from the file. The WisS application loads the data volume via z-slices and we use these slices to update the values in tree leaves that are already initialized to the boundary values (mins to max

possible value and vice versa). During the updating process each leaf (macro cell) that can be affected by the values of the given z-slice is processed using appropriate voxels from the slice. It is important to take into account the voxels laying on the boundaries between macro cells. These voxels are used in more macro cells when detecting the isosurface intersection and thus all adjacent macro cells must be updated with their values.

Inner nodes build The second phase is executed after the volume loading finished and the leaves of the tree are updated according to the loaded data. This phase goes through all inner nodes of the tree from the lowest levels to the root and simply computes the minimal and maximal values of each node according to all node's children minimal and maximal values. When this phase ends, the tree is ready to use.

5.4 Hybrid volume traversal

We traverse the volume in two phases. In the first phase we use the octree acceleration structure for traversal and in the second one we traverse the volume directly voxel by voxel using the three dimensional DDA algorithm.

The tree structure is stored in the texture (see section 5.2.1 on page 33) but since almost whole shared memory is unused we decided to experiment with the shared-memory tree pre-caching. Because the shared memory size per thread block is limited and thus it is not possible to store whole tree there, we pre-cache only a few upper levels of the tree. The tree traversal can be therefore divided into two sub-phases:

- the traversal of the tree in the shared memory and
- the traversal of the rest of the tree in texture.

Both sub-phases use the same traversing algorithm, only the source of the tree data differ.

When the tree traversal reaches a tree leave representing a macro cell the traversal it switches to direct volume traversal phase which processes the content of the macro cell. The depth of the tree is determined by the dataset size and the macro cell size which will be set experimentally to the most appropriate value according to the performance measurements.

5.4.1 Tree pre-caching in the shared memory

The tree pre-caching functionality is optional and can be turned off without leaving any performance impact on the rest of the traversal. This allows us to correctly measure the performance impact on the traversal process when comparing to traversal without using pre-caching.

There are at least two ways of storing the tree in the shared memory possible. The first way is to store array of nodes of the same data type as the original octree. Accessing the nodes in such structure requires the conversion of the original data type to `float` type if we want to use the same traversal code for both the tree shared memory and the tree in texture or it alternatively requires special test for isovalue interval checking. Next, it would require the original tree data (without the `float` normalization) accessible in global memory during shared memory initialization on kernel start. The second way is to store the tree nodes as `float` values as fetched directly from the tree texture. This approach allows us to access the shared memory data without any conversion or special

interval test and data can be initialized directly by copying from the texture. On the other hand it requires more memory for storing the tree. Since we do not expect huge impact of shared memory tree pre-caching on traversal performance, we decided not to spend a lot of time on its implementation and thus we have chosen the second approach for marginal experiments.

We are expecting our implementation of the stackless octree traversal to consume at least 32 registers and only block sizes that it can achieve reasonable multiprocessor occupancy (50 %) for are 64, 96 and 128 threads per block. For these configurations we cannot use more than 2048 B, 3072 B or 4069 B respectively of shared memory to preserve the maximum achievable occupancy. If the depth of the pre-cached tree is set to 3, the number of it's nodes will be $\frac{8^3-1}{7} = 73$ therefore it will occupy $73 \times 2 \times 4 = 584$ B of shared memory. It is evident that it is not possible to store larger tree because increasing the depth by one more level would increase the size of the tree cca 8 times and even for 128-threads configuration there won't be enough memory available.

As stated beforehand, the initialization of the tree in the shared memory is simple – we just read the nodes from the texture and directly save them into the array in the shared memory. To employ the resources the most efficiently the tree copy is distributed among all threads of block and the data are written to shared memory the way that doesn't cause so-called *bank conflict*. Bank conflict occurs when two or more threads are accessing the same block of the shared memory called bank at the same time. In this case the serialization of accesses must be performed by the shared memory management system, and thus the memory accesses are more time-costly. However since each node is accessed by just one thread during the copy and the size of **`float`** min and max values is the same as the shared memory bank size (32 b), there will no bank conflicts occur during the copying process.

The pre-cached tree accessing is simple too. When the tree pre-caching is enabled the tree accessing routine just decides whether the min/max pair will be fetched from the texture or will be read from the shared memory array according to the current node level. This way the branching of the code caused by the tree source switching is moved to the lowest possible level and the added code branches are the shortest possible. When the tree pre-caching is disabled no branches are preformed due to the conditional code compilation, therefore the ability of pre-caching means no performance impact on tree traversal with pre-caching disabled.

5.4.2 Octree traversal

Globals

The octree traversal requires several global variables. First the ray structure variable is used called `eyeRay` of type `RayEx`. It contains the origin of the ray, the direction of the ray and the precomputed inverted ray direction for more efficient ray-plane intersection computations. During whole octree traversal the current node identification is kept up-to-date. The identification consists of two variables:

- `level` – containing the node level (depth in the tree) and
- `levelCoords` – containing the three-dimensional coordinates of the cell represented by the node on the current level.

Next, three **`float`** ray position parameters are kept up-to-date during whole traversal. The ray position parameters are referred to as *t-parameters*. First t-parameter is `tCurrent`, it

represents the current position on the ray. This position is subsequently moved forward during the traversal. Next t-parameter is `tOutCell`. It represents the point where the ray leaves current node and is mainly used when detecting node's children intersected by the ray and when up-traversing from the node. The last global t-parameter is called `tOutGlobal`. This global variable represents the point where the ray exits the volume data and is used for determining whether the tree traversal is behind this point and should terminate. Except the globals needed for the traversal the color buffer named just `color` of type **`float4`** used for color and transparency integration.

Initialization

A few steps are performed during the traversal initialization. First the coordinates on the projection plane for the thread are computed and then remapped to viewport coordinates. Subsequently the ray is initialized such that it points to the point determined by these coordinates. The current node identification and color buffer are initialized to neutral values too. After this basic initialization the ray is intersected with bounding box of the dataset. The *in* and *out* intersections are used for initializing the `tCurrent` and the `tOutGlobal` variables. If there is no ray-box intersection the thread is terminated immediately.

When the initialization phase is done the main traversal cycle can start.

Main cycle

Main traversal cycle (as previously shown in the pseudocode 4.3 on page 29) has two main branches handling two situations that can occur during the tree traversal:

- entering an inner node and
- entering a leaf.

After processing whole an inner node or a leaf, the traversal moves up.

Processing an inner node When processing an inner node we first determine the node's child intersected by the ray on current position. Then we traverse through the children until the child containing the isovalue is found or we are outside the allowed range.

Locating the current child When entering the node we search for currently intersected child via intersecting node's mid-planes. The intersecting of the mid-planes yields points where the ray traverses from one intersected child to the next intersected child of the node. We intersect all mid-planes of the node and find the closest intersection that is after the current ray position (we might have processed some children already), before the `tOutGlobal` t-parameter (octree may be larger than the dataset) and before the `tOutCell` – the *out* t-parameter of cell represented by current node (intersections behind this value are invalid). After finding the closest next mid-plane intersection we have point inside the current child determined by the `tCurrent` variable and point where the ray leaves the child determined by the found closest intersection (stored in variable `childTOutCell`). From this two point we can compute coordinates of the current child.

Having the two points inside the cell we first simply compute the t-parameter of the point laying in the center between the two points. This point is guaranteed to be inside

the cell since it is convex. If we used only one point (e.g. the point where the ray leaves the cell) it wouldn't be clear which cell it would belong to. Now we compute the full coordinates of the point from to the ray and the computed t-parameter. The level coordinates of the child containing the point are computed simply by shifting the full integer coordinates of the point according to the level of the child.

The functionality of computing the coordinates from two t-parameters is implemented in function `GetNodeFromT()` and whole current child locating is implemented in function `EnterNodeSL()`.

Searching for non-empty child After the current child is identified it must be checked for being non-empty (i.e. for containing the current isovalue). To perform this test the two-dimensional node index must be computed via the linear index (see the section 5.2.1 on page 33). Then the node's min/max pair must be fetched from the octree and finally it is checked whether the isovalue lies inside the interval defined by the pair. If it is true the node can possibly contain an isosurface and have to be processed otherwise it cannot and will be skipped. Note that this is the only step of the tree traversal algorithm which touch the tree data structure.

If the child is empty the traversal algorithm must find the next child and move to it. Searching for the next child is very similar to searching for the current child in the previous step. First the current position (i.e. the current t-parameter) is moved to the point where ray leaves the current child (i.e. the current child cell out t-parameter – `childTOutCell`) then the mid-planes coordinates are computed and intersected with the ray one after the other. The `childTOutCell` is then updated to the closest next intersection with mid-planes and the new child level coordinates are computed using the `GetNodeFromT()` function. After moving the current t-parameter to child's out t-parameter we must do two checks. First, we need to check whether we haven't reached the back side of the cell by comparing the current t-parameter and child's out parameter. Second, we also need to check whether we are still inside the dataset spatial range by comparing the current t-parameter to global out t-parameter. When comparing the t-parameters we always use an *epsilon-adjusted comparison* to avoid floating-point inaccuracy induced errors. If the non-empty child is found the traversal moves the current node identification (level and level coordinates) to this child otherwise the traversal moves upwards to the node's parent.

The functionality of searching for the first non-empty child from the previously found current child is implemented in the function `MoveToFirstNonEmptyChildSL()`.

Infinite cycle problem During the process of searching for the first non-empty child a problem can occur for some mutual positions of the ray and the node's mid-planes. When the algorithm decides to skip the current child because of its emptiness it searches for the closest next mid-plane intersection. If the ray is almost parallel to the mid-plane that is intersected by the ray when leaving the current child, the next intersection point can be very close in one dimension to the intersection point where the ray leaves the current child (see the figure 5.3 on the following page). If they are very close the inaccuracies caused by float-point type precision can occur that can lead to incorrect next child computation. In some cases it can result in situation when the current child is detected as the next child. In such a situation the algorithm "moves" to that child but de-facto it stays in the same child and the current t-parameter stays untouched. After this the algorithm continues processing the already processed child and starts to loop in the infinite cycle.

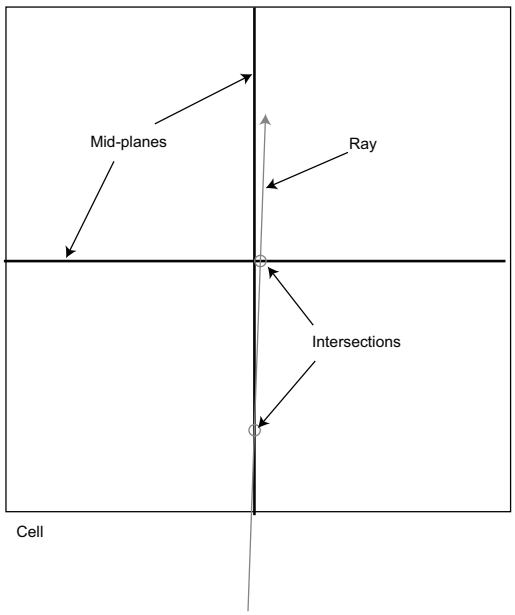


Figure 5.3: Infinite cycle problem configuration. If the intersections are very close in at least one dimension, float-point computation errors can occur.

We decided to solve this problem by adding the test detecting this situation right next to the "is empty" test. This test detects whether the current and child out t-parameters are far enough from each other in each dimension. Unfortunately this test is quite costly. It requires to compute the same point between the current and the out point as is used in the process of computing the child coordinates from t-parameters, then it computes one vector difference between this point and the center point of the node and finally it compares the distances in each dimension with hardwired safety threshold. Moreover, since the voxel size can be optional in each dimension the threshold must be adjusted for each dimension according to the voxel extent. The threshold for each dimension is stored in the constant memory.

The problem of this solution is that it skips some macro cells that would produce isosurface intersection if we processed them. This leads to some missing-surface artifacts as shown in figure 5.4 on the next page. The threshold is set experimentally to value that produces a reasonable trade off between the image errors and the immunity against the infinite cycle problem.

Moving upwards After processing a node the algorithm moves up to the node's parent. First the current t-parameter is moved to the out t-parameter of the node. The new current t-parameter value must be checked as during searching for the next non-empty child for being inside the dataset spatial range. This is done via the epsilon-adjusted comparison of the updated current t-parameter to the global out t-parameter. After moving the current position the level is decreased by one, the current level coordinates are shifted by one position to get the coordinates of the parent, and finally the new out t-parameter for the new current node must be computed.

The out t-parameter is computed as the closest intersection of the ray with back-sides of a cell represented by the node but since we generally don't know what sides are the "back" ones, we need to intersect all six sides. This is done by intersecting three pairs of parallel planes defined by pairs of opposite sides of the cell, determining maximums of each pair intersections (this yields all possible back side intersections) and then picking

Figure 5.4: Artifacts caused by skipped children when searching for next non-empty child. In the left image horizontal line is visible. In the right image there are lots of transparent points in the isosurface. Both images are zoomed using nearest neighbor filter to preserve the sharpness of artifacts.

the minimum from the pair maximums. This way we get the t-parameter describing the point where the ray leaves the cell.

The process of moving upwards should continue while the current t-parameter is on the back side of the current cell. However instead of iterating whole cycle inside the implementing routine we decided to do only one step up each time the routine is called and routine is called repeatedly once per one main traversal cycle until the end condition is not satisfied. This slightly improves the branching behavior because if there are more threads that need to traverse upside the tree but they don't need it exactly the same time, there is still a chance to have their execution paths partially converged (i.e. if they start up traversing closely one after another) instead of executing separate whole loops.

This functionality is implemented in function `MoveUpwardsSL()`.

Processing a leaf When a octree leaf is reached the grid traversal routine (`IntersectGrid()`) is called. It prepares the parameters for adjusted grid traversal implementation from original WisS application that is then called. The macro-cell is then traversed by it using the 3D DDA algorithm with isosurface detection and when the isosurface is detected the shading is performed and the color buffer is updated with result. The grid traversal uses early-ray termination optimization technique that stops the traversal if the buffer's alpha exceeds hardwired constant (`EARLY_RAY_TERMINATION_ALPHA`). The tree traversal stops when the grid traversal stop condition is satisfied too.

Chapter 6

Results

After implementing of our acceleration extension of the WisS application we have performed several performance test to determine its behavior. We used desktop computer with GeForce 200 series GPU with 1.3 computing capability. The computer specifications was following:

- Processor: Core 2 Quad 64-bit, 2.4 GHz
- Operating memory: 8 GB
- Operating system: Windows XP, Professional x64 Edition
- GPU: NVIDIA GTX 280 (1024 MB, 30 multiprocessors, 602 / 1296 MHz)

For testing purposes we used several medical and industrial CT-scan datasets with various data characteristics. Some of them are generating large isosurfaces filling most of the dataset range, some of them contain very sparse isosurfaces. Datasets also vary in size and complexity of surfaces generated what allowed us to study acceleration behavior in various conditions.

Although our implementation is capable to handle transparent isosurfaces we are focused mainly on non-transparent isosurfaces. Transparency can cause that more complex isosurfaces are visible with worse ratio of space occupied by visible non-empty macro cells to amount of visible empty space that can be skipped by the acceleration algorithm. Empty space skipping acceleration structures thus tend to behave worse in such conditions.

In accelerated code we use grid traversal with as little changes as possible to get more purposeful results when comparing the non-accelerated to the accelerated code rendering performance. This means that accelerated isosurface rendering kernel uses grid traversal that supports more parallel datasets although the tree traversal always traverses just one.

All tests were performed in rendering window sized to 512×512.

6.1 Tuning the configuration

As there are lots of parameters affecting the final performance of the rendering kernel we decided to first experiment with them and do the final measurements just after the system is tuned for the best behavior.

We tested various configurations with the so-called *exact isosurface test* enabled instead of the so-called *linear test*. The exact test detect the isosurface analytically and

Dataset	Macro Cell Levels	FPS
Abdominal	2	22
$512 \times 512 \times 174$	3	19
	4	16
Engine head	2	26
$1004 \times 525 \times 439$	3	24
	4	20

Table 6.1: Performance for various macro cell sizes.

thus precisely but at higher computation cost. Second test available – the linear test, is a lot quicker than the exact test but it produces isosurfaces with less fidelity damaged by topological artifacts. We decided to optimize parameters with the exact test because we are interested in acceleration algorithm behavior when used with more time-consuming isosurface computations. This approach gives us better estimation on behavior when used with more sophisticated methods like better shading methods or secondary ray techniques. We couldn't afford to experiment with both tests because the testing computer with GTX GPU was not our property and it was only borrowed to us for some time and the measurements were relatively time-consuming. For the same reasons we only did all tuning experiments for small sets of volume data only. Moreover we have had preliminary testing done for most of experiments on older and slower GPU with computing capability 1.1 that gave us basic idea of future final behavior and final experiments and measurements were only needed to confirm it.

All experiments are based on average fps achieved on each select data sets. Fps measurements were done using a WisS application built-in fps testing feature. It rotates the currently loaded dataset around the y-axis in 360 steps by 1 degree per step and computes computes the minimum, maximum and average frames count per second.

6.1.1 Macro cell size

First we decided to inspect whether the macro cell size that is unambiguously better the others for various datasets. Preliminary measurements had shown that the smaller macro cells are the better results are achieved. The final measurements confirmed this behavior as shown in table 6.1.

During the pre-implementation analysis of probable behavior we ended up expecting that there will be some trade-off between the shallowness of the tree and the overhead of macro cell traversal initialization and the optimal macro cell size will lie somewhere "in the middle". The explanation of this expectation is following. The smaller the macro cells are the more empty space is the tree traversal able to skip and save the grid traversal. In other words, the smaller macro-cells are the better "tightness" is achieved by the macro-cell bounding volume and less empty voxels need to be traversed by the DDA traversal. To prove this we performed internal measurements of the volume traversal algorithm. It shown us that decreasing the macro cell size from four to two level saves in average about 70 % of traversed voxels. On the other hand the smaller macro-cells are the more macro cells are in the tree and more times the macro cell traversal must be initialized. Although it may seem that the macro cells count is inversely proportional to macro cell size, the macro cells count is smaller that one can expect. This is because the smaller macro cells don't fill full space of larger macro cell – they just better approximate the isosurface. This has been proven indirectly by another internal measurement that investigated amount of

Dataset	Registers	Occupancy	Local memory	FPS
Plastic plug	64	25	92+32	13
$553 \times 768 \times 366$	32	50	320+32	14
	24	63	288+32	14
	20	75	328+32	14
	16	100	464+32	12
Med4	64	25	92+32	17
$512 \times 512 \times 646$	32	50	320+32	19
	24	63	288+32	16
	20	75	328+32	16
	16	100	464+32	14

Table 6.2: Performance for various registers usage limitations.

time that macro cells traversal spends on its initialization. Although the times measured are quite deformed (e.g. because of threads time slicing) it tells us that initialization takes only small portion of time spend for grid traversal even for very small macro cells (2^3).

Small macro cells has proven to be more computation efficient but unfortunately produce very large trees. We decided to use the smallest reasonable macro cell size of two levels – 2^3 for all next measurements.

6.1.2 Registers count

Next after setting the macro cell size we focused on registers consumption. Discovering that our implementation of tree traversal with exact test DDA isosurface detection requires more than 64 registers (using current CUDA 2.1 compiler) was a bit disappointing for us. Such an amount of registers results in very low multiprocessor occupancy. However we are suspecting the CUDA compiler of sub-optimal registers management.

We decided to experiment with compiler option `maxregcount` that sets the maximum amount of register that a thread can use. Less used variables that do not fit into registers are then temporarily stored in local memory and loaded into registers only when they are needed. This is called *register spilling*. Decreasing the number of usable registers can improve multiprocessor occupancy but on the other hand it requires more accesses to slow local memory. We experimented with various registers counts to find the optimal registers level and results are shown in table 6.2 where the resulting occupancy and used local memory for each register count is also added.

As can be seen, the performance peak is reached when using 32 registers per thread. This again confirmed our preliminary results. However interesting paradox can be seen in local memory usage. The local memory grows as the allowed registers count decreases as we expected except the 32-registers case. This indirectly indicates that registers management doesn't work very properly.

We use 32-registers restriction for all next measurements.

6.1.3 Threads per block and warp shape

Next area we decided to examine is the influence of the threads count per block and warp shape on performance of our acceleration algorithm.

We tested only three different block sizes on one dataset due to same reasons as in previous measurements. We tested only the multiples of 64 because they are recommended

Dataset	Threads per Block	FPS
Med4	64	17
$512 \times 512 \times 646$	128	18
	192	14
	256	16

Table 6.3: Performance for various threads count per block of threads.

Dataset	Warp shape	FPS
Plastic plug	32×1	18
$553 \times 768 \times 366$	16×2	16
	8×4	17
Med4	32×1	21
$512 \times 512 \times 646$	16×2	19
	8×4	19
Abdominal	32×1	20
$512 \times 512 \times 174$	16×2	19
	8×4	20

Table 6.4: Performance for various warp shapes.

for their best behavior in avoiding register memory bank conflicts[3]. As shown in table 6.3, 128 threads per block did the best and this is the block size that we have chosen for next measurements.

Warp shape can slightly affect performance due to execution paths coherency issues. We expected that warps with the most compact shape (the closest possible to square) will cause that threads in the warp will be the closest to each other and thus they will process more correlated data while non-compact warps will process data that are wider and thus less correlated. We did a few measurements on two datasets with three possible warp shapes (shape of warp is determined by the size of a block) but results were unclear therefore we decided to add one more dataset for sureness. The results are shown in table 6.4. The most compact and the most non-compact (line) shapes contest together but there is no obvious winner. The shape of warp doesn't affect the performance very significantly in our case. Although the line-shaped warp did a little better, the results are very similar for all shapes and we decided to choose the compact-shaped warp according to the data-correlation heuristics for future measurements.

6.1.4 Pre-cached tree

At the end of the system configuration tuning we also measured the impact of using the tree pre-caching in the shared memory on the rendering performance. The results are shown in table 6.5 on the following page. As during the preliminary testing on older GPU, using the shared memory pre-caching brings no performance improvement. Probable explanation of this behavior is that the shared memory access speed doesn't cover the pre-caching mechanics overhead, i.e. copying the tree into the shared memory and switching between the texture and the shared memory when accessing the tree.

Dataset	Tree FPS	Pre-cached Tree FPS	Performance Gain
Abdominal $512 \times 512 \times 174$	20	20	0 %
Engine head $1004 \times 525 \times 439$	28	28	0 %

Table 6.5: Performance gain for tree pre-cached in shared memory.

Dataset	Isosurface Test	DDA FPS	Hybrid Tree FPS	Gain
Plastic plug	Linear	38	36	-5 %
$553 \times 768 \times 366$	Exact	12	14	17 %
Mouse skeleton	Linear	57	80	40 %
$201 \times 201 \times 326$	Exact	18	37	106 %
Casting	Linear	26	47	81 %
$441 \times 837 \times 411$	Exact	8	20	150 %
Abdominal	Linear	28	53	89 %
$512 \times 512 \times 174$	Exact	8	21	163 %
Engine head	Linear	35	56	60 %
$1004 \times 525 \times 439$	Exact	9	27	200 %
Med4	Linear	16	43	169 %
$512 \times 512 \times 646$	Exact	5	18	260 %

Table 6.6: Performance gain of accelerated isosurface renderer.

6.2 Performance gain

Finally, after we tuned the system settings, we performed set of final performance measurements. We measured the rendering performance of accelerated isosurface rendering with both linear and exact isosurface tests and then we compared it to the original WisS isosurface rendering performance. This way we can determine the ability of our accelerating structure to improve the performance of renderer. Results are show in table 6.6 sorted in ascending order according to performance gain on exact test rendering.

We measured the performance on datasets with various characteristics and so are the results. The *Plastic plug* dataset (see image 6.1) is relatively large and this is the characteristics that generally makes the empty-space skipping accelerators behave well. However isosurface in this data set is pretty complex and it fills the space very densely what requires lots of macro cells to be processed and allows only a small portion of empty space to be skipped by the tree traversal. The result is that we achieve only 17 % speed-up for exact test isosurfacing and even small performance loss for linear test.

The *Mouse skeleton* dataset (see image 6.2) is accelerated two times on exact test. Though it is relatively small dataset the isosurface rendered is very sparse, and therefore there is a lot of empty space that can be skipped.

Next three datasets were rendered at least two and a half-times faster with exact test when our acceleration structure was used. Dataset named *Casting* (see image 6.3), containing an industrial CT scan of some component part did well despite of not having very sparse isosurface. We explain it by the combination of two factors: there is relevant amount of empty space to skip and the surface is quite topologically simple (non-complex) that requires less macro cells to cover the surface. Isovalue for dataset *Abdominal* (see image 6.4) containing the CT scan of a human body is set to value that visualize mostly

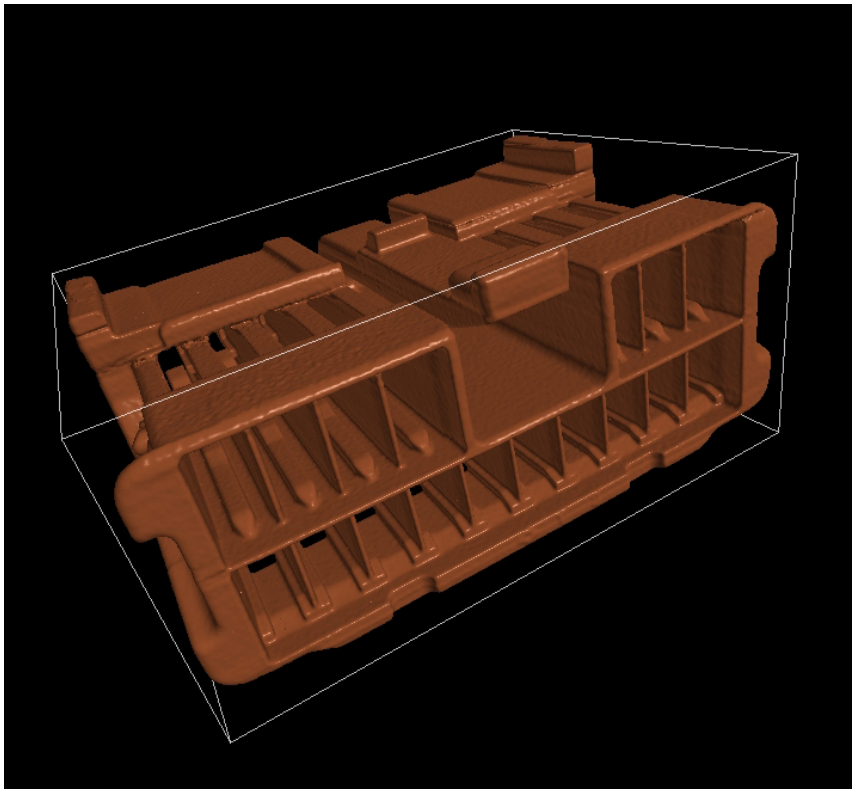


Figure 6.1: The *Plastic plug* dataset.

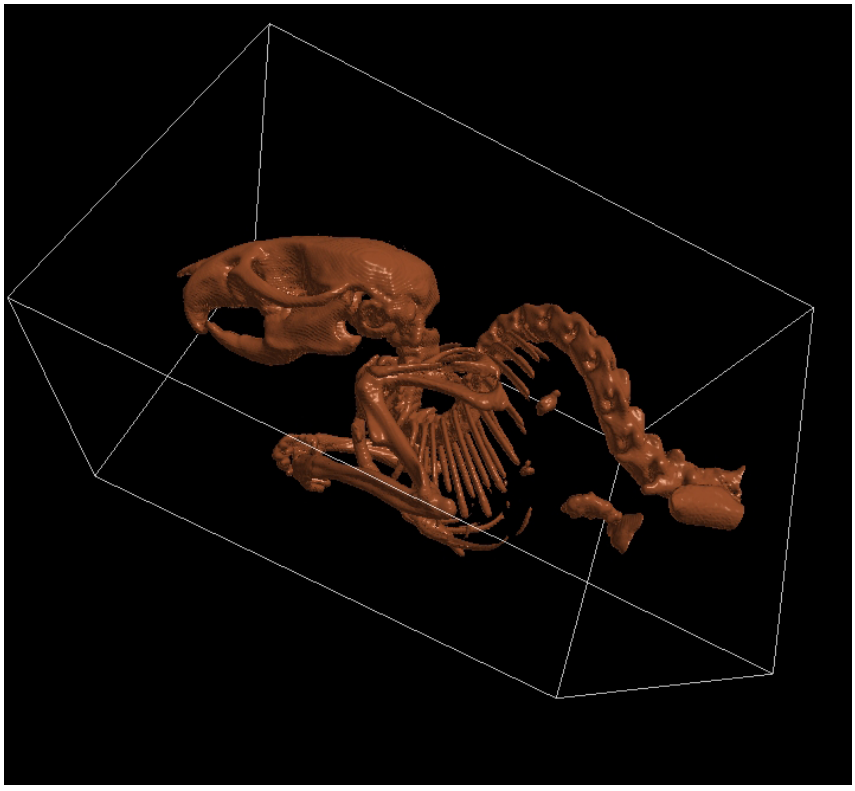


Figure 6.2: The *Mouse skeleton* dataset.

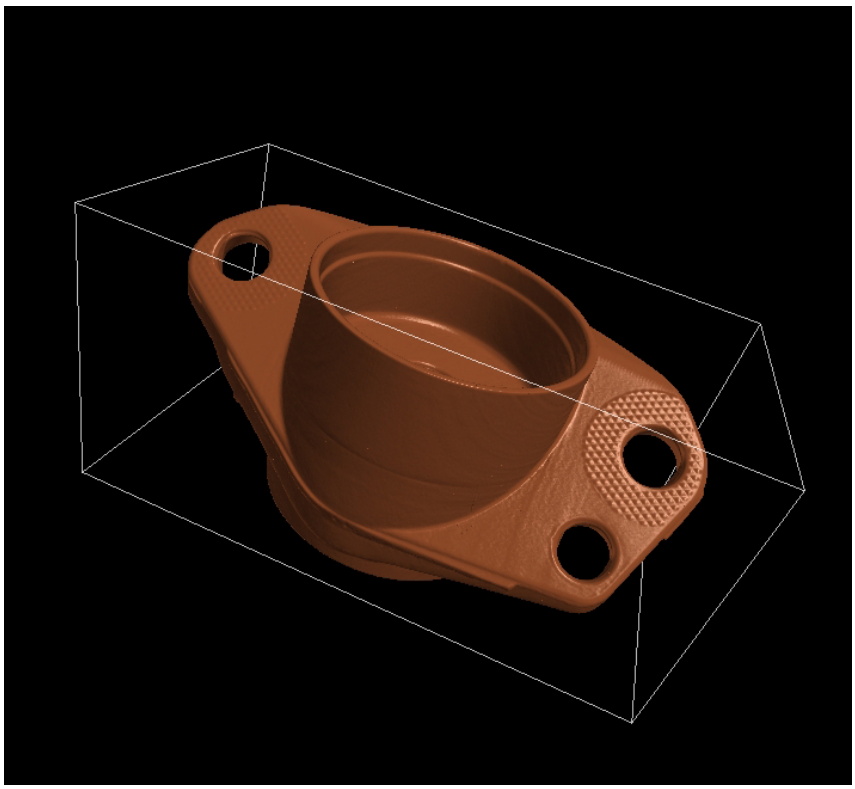


Figure 6.3: The *Casting* dataset.

bones and similar tissues. The resulting isosurface is topologically complex but the amount of empty space that can be skipped affect the speed-up more significantly. Performance on dataset named *Engine head* (see image 6.5) has taken us by surprise. Large part of the isosurface that we were interested in is situated near the boundary of volume, and thus the direct voxel traversal requires only a few steps to reach the isosurface. This makes the ratio of voxel traversal steps to tree traversal steps less advantageous for the tree approach. Fortunately there are two compact empty spaces with considerable size at the end of the shape that probably allows tree acceleration to get the jump on voxel traversal.

We achieved the best speed-up results on dataset named *Med4* containing large human skull CT scan. The speed of rendering increased more than three and a half-times. Probable explanation of such behavior is similar to the *Casting* dataset rendering – the isosurface of skull is not very rugged that requires less macro cells to bound the surface and there is plenty of space between the skull surface and the volume boundary that gets skipped during the volume traversal.

Datasets *Casting*, *Mouse skeleton* and *Engine head* were made by the Volume Graphics company (`http://www.volumegraphics.com`). The dataset *Med4* was scanned by Department of Anthropology at Faculty of Science of Charles University in Prague.

6.3 Comparison to other renderers

We compared our result with another GPU-based isosurface renderer developed at Saarland University by Maršálek et al. [18]. This renderer uses the direct fixed-step ray marching approach for volume traversal with linear isosurface detection test. The traversal step was set to size of one voxel.

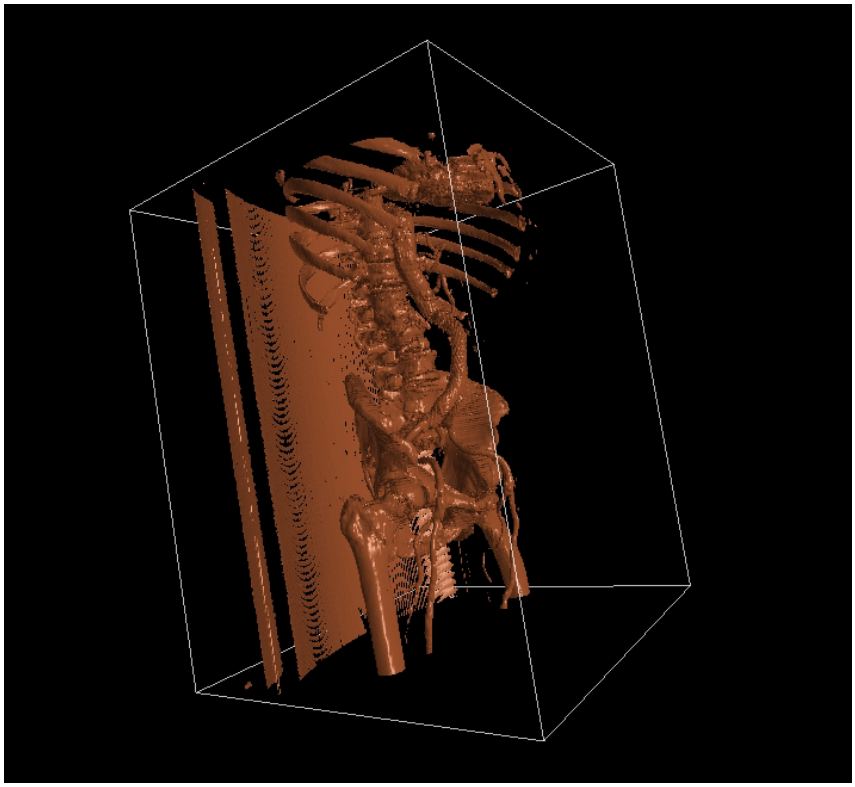


Figure 6.4: The *Abdominal* dataset.

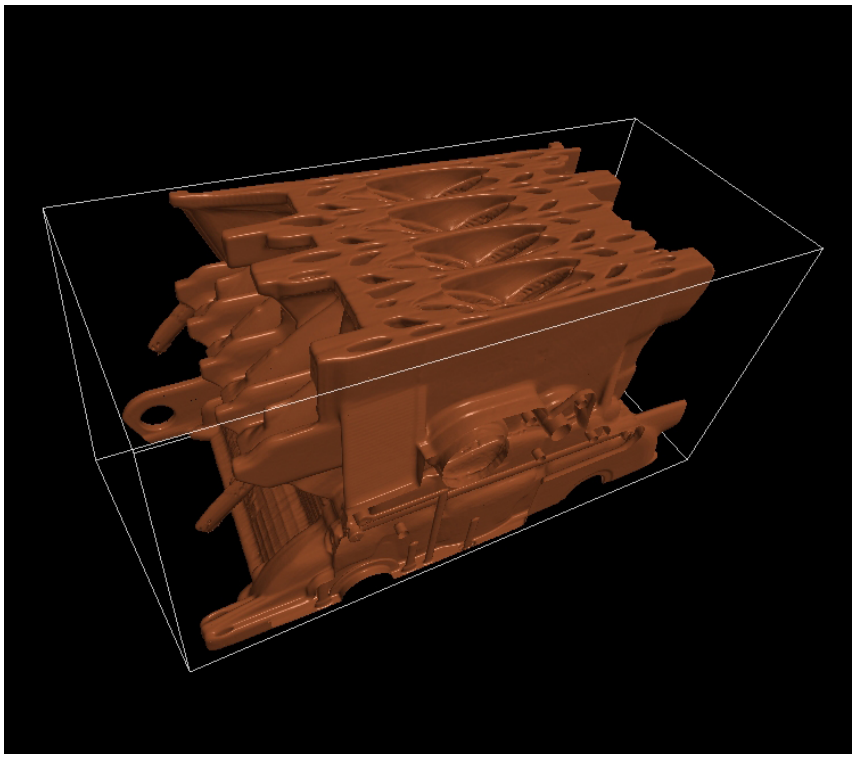


Figure 6.5: The *Engine head* dataset.

When compared on the same datasets as were used in our main test, the ray marcher is about 2.2-times faster than original WisS DDA-based isosurfacing, and therefore it was faster than WisS linear isosurfacing with octree acceleration in most cases. However it is somehow hard to compare these renderers since the ray marcher uses the volume marching instead of the DDA traversal algorithm used in the WisS application. The difference is that the fixed-step volume traversal can be implemented more efficiently than the DDA algorithm and larger step step can be used for faster traversing. The problem of fixed-step volume traversal is that the larger the step is the more inaccuracies and artifacts the traversal produces while the DDA traversal guarantees that no visible isosurface point gets skipped. Moreover the ray marcher is highly optimized using low-level techniques like by-hand registers management that results in lower register consumption (12 live registers) and thus higher multiprocessor occupancy (83 %). Last but not least, it is important to remind that WisS isosurfacing code still supports two parallel datasets that makes implementation slightly more complicated and less efficient than one-dataset variant.

We haven't found any other GPU-based isosurface renderer comparable to our application because all renderers implement some kind of linear or even simpler isosurface detection test with less topologically accurate isosurfaces.

Chapter 7

Conclusion

In our work we investigate the acceleration possibilities of ray tracing-based isosurfacing implemented on consumer GPUs.

In the first chapter of this thesis we made a brief introduction to volume rendering, isosurfacing and GPU utilization for volume rendering purposes. First the computer graphics and data visualization is introduced. Then the volume rendering basic principles are explained with the emphasis on isosurface visualization. After the introduction the goal of the thesis is set – to propose, implement and test an acceleration technique for GPU isosurfacing.

In the next chapter we reviewed the previous works somehow connected to the topic. We addressed classical isosurfacing approaches and discussed few work on GPU-based isosurfacing. Since the GPU-based isosurfacing is not yet satisfactorily covered we then focused on general ray tracing works on GPU because we expected that they needed to solve similar problems that we would. After the review we described the main idea of our approach – octree-based volume traversal.

Next we described current GPU architecture and discussed CUDA programming specifics. We then proposed a set of possible traversal approaches that could be used on GPU and discussed on applicability of them. The discussion lead to selection of three traversal approaches that best suited the architecture requirements (mostly the stack memory size usage).

In the next chapter we did more detailed analysis on three traversal approaches select in previous chapter. We provided a comparison and discussed the different requirements of all three traversals. We focused on memory and register consumption as well as amount computation estimation. After considering advantages and disadvantages of the approaches we have chosen the stackless traversal as the implementation candidate.

After the theoretical analysis the implementation phase started. We implemented our acceleration technique as the extension of the Vladimír Hrinčár's WisS application. We designed and implemented linear octree storage in texture, the tree pre-caching in shared memory and hybrid volume traversal. Hybrid traversal first traverses the volume using the octree stored in two-dimensional texture, optionally first levels of the tree can be pre-cached in and loaded from the fast shared memory and when the leaves of the tree are reached the corresponding macro-cell is traversed using slightly modified WisS grid traversal with isosurface detection.

After the implementation phase we performed experiments where we searched for configuration that provides best rendering performance and finally preformed rendering performance comparison of original WisS isosurface rendering and our accelerated implementation. We presented the tuning process and comparison results in the Results

chapter of this thesis. We also compared the performance to another isosurface renderer.

7.1 Work summary

In this work we discussed the possibilities of octree-based acceleration of isosurfacing on GPU architecture. We successfully designed and implemented stackless octree traversal technique on GPU. The results of the work confirmed the hypothetical ability of octrees to increase the rendering performance of ray tracing-based isosurfacing implemented on consumer GPUs.

The work also pointed out a potential problems of the GPU tree acceleration. Since we didn't use any memory for tree traversal stack the shared memory usage was not the problem but the registers usage appeared to be the bottleneck. Also the tree size and the floating-point precision traversal problems need to be addressed.

Possible improvements and future work are discussed in more detail in the next section.

7.2 Future work

Although we presented a functional acceleration technique in this work there are lots of issues that need to be addressed in future works and we believe that performance of the isosurface renderer can be improved even more.

- First the registers consumption needs to be reduced either by reimplementing the tree traversal more register efficiently or by manual register management as used in the ray marcher from [18]. This might allow using stronger register usage restrictions without intensive register spilling into the local memory in order to increase multiprocessor occupancy and thus improving the overall kernel performance.

- The infinite cycle problem during the search for first non-empty child needs to be reviewed to minimize or even eliminate the problem.

- The tree structure used in our implementation was quite memory wasting (e.g. for irregular-shaped datasets). In some cases the octree requires more than two-times larger amount of memory than the dataset that the tree was built for. More memory-saving octree structure should be proposed to solve this issue.

- Tree build process can be pretty slow for large trees and the possibility of using GPU for this purpose should be explored.

- Because pre-caching of the tree in the shared memory turned out ineffective we do not recommend using it in form we implemented it, there is almost whole shared memory free and can be used for other speed-up techniques. For instance more sophisticated tree pre-caching or using the shared memory for some variables that might help minimize registers consumption of kernel.

- The last but not least, our work should be implemented using OpenCL framework to get rid of dependency on NVIDIA GPUs.

Bibliography

[1] Steven Parker Aaron Knoll, Ingo Wald and Charles Hansen. Interactive isosurface ray tracing of large octree volumes. 2003.

[2] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[3] NVIDIA Corporation. Nvidia cuda compute unified device architecture, programming guide, version 2.0, 2008.

[4] Manfred Ernst, Christian Vogelgsang, and Günther Greiner. Stack implementation on programmable graphics hardware. In *VMV*, pages 255–262, 2004.

[5] A Knoll et al. Ray traversal of octree point clouds on the gpu. 2003.

[6] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM.

[7] Johannes Günther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, pages 113–118, September 2007.

[8] Wei Hong, Feng Qiu, and A. Kaufman. Gpu-based object-order ray-casting for large datasets. In *Volume Graphics, 2005. Fourth International Workshop on*, pages 177–240, 2005.

[9] Vladimír Hrinčár. Volume visualization of human skulls. Master's thesis, Charles University in Prague, Prague, Czech Republic, 2008.

[10] James T. Kajiya and Brian P Von Herzen. Ray tracing volume densities. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 165–174, New York, NY, USA, 1984. ACM.

[11] John Kloetzli, Marc Olano, and Penny Rheingans. Interactive volume isosurface rendering using bt volumes. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 45–52, New York, NY, USA, 2008. ACM.

[12] Aaron Knoll, Younis Hijazi, Andrew Kensler, Mathias Schott, Charles Hansen, and Hans Hagen. Fast ray tracing of arbitrary implicit surfaces with interval and affine arithmetic. 2008.

[13] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, Washington, DC, USA, 2003. IEEE Computer Society.

[14] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[15] Marc Levoy. Efficient ray tracing of volume data. *ACM Trans. Graph.*, 9(3):245–261, 1990.

[16] Sukhyun Lim and Byeong-Seok Shin. A half-skewed octree for volume ray casting. *IEICE Transactions*, 90-D(7):1085–1091, 2007.

[17] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, 1987.

[18] Armin Hauber Lukas Marsalek and Philipp Slusallek. High-Speed Volume Ray Casting with CUDA. In *Poster, IEEE/Eurographics Symposium on Interactive Ray Tracing 2008*, August 2008.

[19] Lukáš Maršálek. Osvětlení na gpu. Master's thesis, Charles University in Prague, Prague, Czech Republic, 2005.

[20] Gregory M. Nielson and Bernd Hamann. The asymptotic decider: resolving the ambiguity in marching cubes. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 83–91, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[21] Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive ray tracing. In *I3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 119–126, New York, NY, USA, 1999. ACM.

[22] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive ray tracing for isosurface rendering. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 233–238, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[23] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. (Proceedings of Eurographics).

[24] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[25] Timothy John Purcell. *Ray tracing on a stream processor*. PhD thesis, Stanford, CA, USA, 2004. Adviser-Hanrahan„ Patrick M.

[26] Christof Rezk-Salama, Markus Hadwiger, Timo Ropinski, and Patric Ljung. Advanced illumination techniques for gpu-based volume ray-casting. In *Course at ACM Siggraph Asia*, 2008.

[27] Niels Thrane and Lars Ole Simonsen. A comparison of acceleration structures for gpu assisted ray tracing. Master's thesis, University of Aarhus, Denmark, 2005.

[28] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination.* PhD thesis, Computer Graphics Group, Saarland University, 2004.

[29] Ingo Wald, Heiko Friedrich, Gerd Marmitt, and Hans-Peter Seidel. Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–572, 2005. Member-Philipp Slusallek.

[30] Ingo Wald, Timothy J. Purcell, Joerg Schmittler, Carsten Benthin, and Philipp Slusallek. Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports*, 2003.

[31] Wikipedia. Graphics processing unit — wikipedia, the free encyclopedia, 2008. [Online; accessed 11-December-2008].

[32] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Trans. Graph.*, 11(3):201–227, 1992.

Appendix A

User manual

A.1 Running the application

Precompiled 32-bit version of WisS application is shipped with the thesis on the DVD (contents of the disk is summarized in the appendix B on page 58). To run the application on a PC with Microsoft Windows XP or Microsoft Windows Vista operating system (32-bit or 64-bit versions) the CUDA-enabled NVIDIA graphics card is required. Using the latest stable NVIDIA GPU driver is recommended but the driver released not before the 2009 (CUDA version 2.1) is required for application to work properly. .NET runtime version 3.5 is required for running too.

The application can also be built directly from the source code provided on the DVD (see appendix B for details). A Microsoft Visual Studio 8.0 solution is provided that allows to build the application more conveniently. To build the application successfully the NVIDIA CUDA SDK version 2.1 or newer and NVIDIA CUDA toolkit version 2.1 or newer must be installed. Both can be downloaded from `http://www.nvidia.com/cuda`. When the provided Visual Studio solution is used for the 32-bit or 64-bit architecture the output files are stored in directory `testing` or `testing64` respectively, situated in the root directory of the source code directory.

To run the application run the `WisS.exe` file.

A.2 Application GUI

The GUI of the WisS application is shown in the figure A.1 on the next page.

Basic tools Three basic tools are provided:

- BBox – toggles whether the the bounding box is displayed or not.
- Ray Info – enables gathering information about volume along the ray (available in DVR mode only).
- FPS Test – runs the performance test.

Dataset properties Allow setting the size, the voxel extent and the bit depth for the dataset to be loaded.

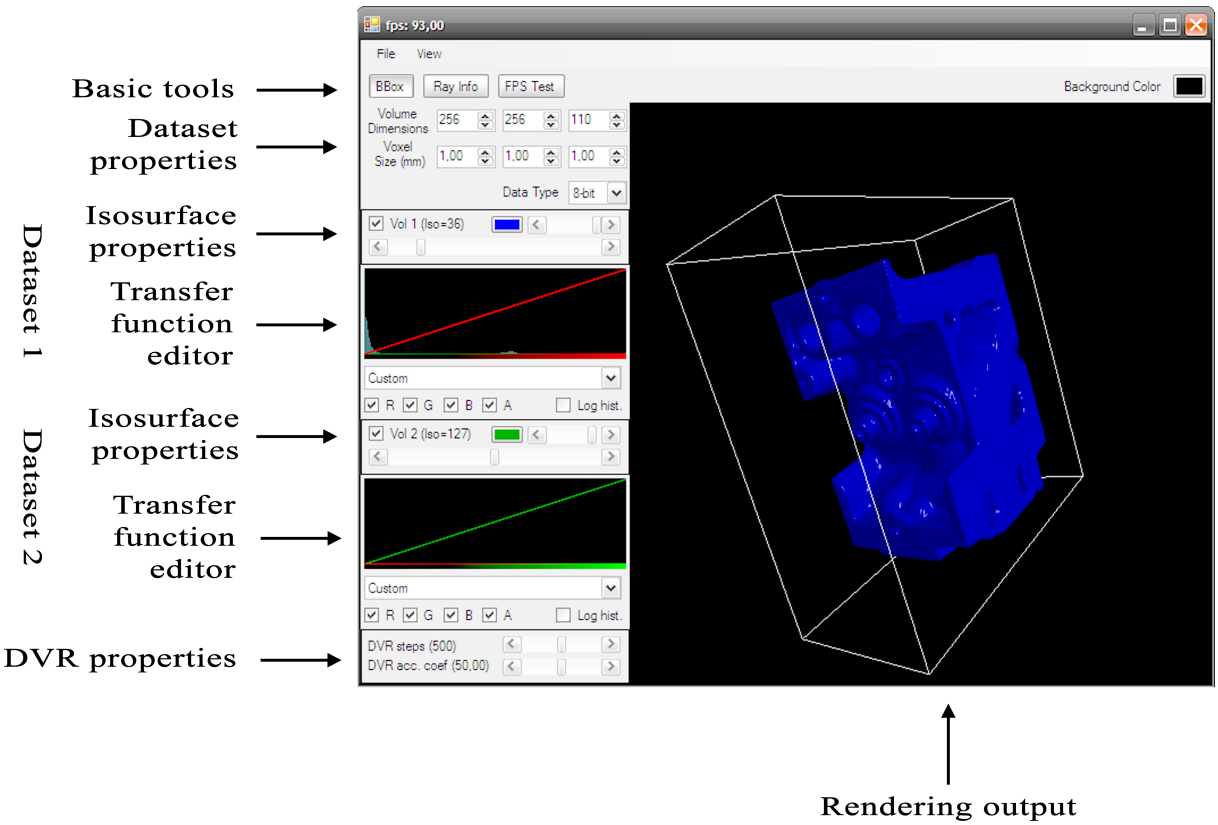


Figure A.1: The WisS application user interface.

Isosurface properties Allows enabling and disabling the first of the second dataset and setting their color. The scrollbar on the right allows adjusting transparency of the isosurface. The scrollbar under the transparency scrollbar and enabling check box allows to adjust the isovalue for a dataset.

Transfer function editor It allows edit a transfer function for a dataset (used in the DVR mode only)

DVR properties Allow to adjust behavior of DVR rendering mode.

A.3 Brief tutorial

The WisS application is able to read the raw data only. Before loading a raw file (using the main application menu) the dataset properties (dimensions, voxel size and bit depth) have to be set manually. Since this can be pretty annoying for user, we added simple volume header functionality that allows to load the dataset with settings stored in another file. The file must have the same name as the appropriate raw data file with the "`.vh`" suffix. If the volume header file is loaded the settings are read from it and appropriate dataset file is then loaded automatically if it is available in the same directory. The volume header file contains several ordered space-separated values whose meaning is explained in table A.1 on the following page.

After a successful load of a dataset required rendering mode needs to be chosen in the View sub-menu in the main application menu. The rendering mode implemented in the work is called Transparent Isosurfaces (octree accelerated).

Value number	Meaning of the value	Data type [range]
1	X dimension of the dataset	integer
2	Y dimension of the dataset	integer
3	Z dimension of the dataset	integer
4	Bit depth of the data	integer
5	The size of a pixel in x dimension	float
6	The size of a pixel in y dimension	float
7	The size of a pixel in z dimension	float
8	Unused value	–
9	Isosurface alpha value	float [0.0, 1.0]
10	Isovalue	float [0.0, 1.0]

Table A.1: The content of the volume header file.

After the load the dataset if rendered in the rendering window according to current settings – isovalue, transparency and isosurface color. These settings can be adjusted on the fly by user.

Testing data are available on the provided DVD as described in appendix B on the next page. If data are too large for loading to the current GPU a warning is shown, however from time to time the CUDA API doesn't signalize allocation problems, and because of this the application crashes. We are not able to solve this problem since we don't know what is causing it. We noticed that unexpected lack of memory can be sometimes caused by two-display PC setup (disabling one display can free some GPU memory) and applications that use large image buffers thus avoiding these conditions can solve some memory problems.

Appendix B

Contents of the DVD

bin Executable version of the WisS application for 32-bit architecture. Execute the `WisS.exe` binary to run the application.

data Example raw datasets.

doc Two electronic versions of this thesis. One for electronic viewing and one for printing (without colored links).

src This directory contains the solution file for building the application in Microsoft Visual Studio 8.0 and directories with files necessary for building the application.

- **CULib** Renderer library.
  - **lib** Support libraries.
  - **src** The the library source code files.
- **Properties** .NET files.
- **src** Source code files of the core of the application.
- **testing** An output directory for a 32-bit compilation used for testing.
- **testing64** An output directory for a 64-bit compilation used for testing.

Index